

**STRICT: A language and tool set
for the design of
Very Large Scale Integrated Circuits**

PhD Thesis

by Albert Koelmans

**Department of Computing Science,
University of Newcastle upon Tyne,
March, 1995.**

NEWCASTLE UNIVERSITY LIBRARY

096 51068 6

Thesis L5797

ABSTRACT

An essential requirement for the design of large VLSI circuits is a design methodology which would allow the designer to overcome the complexity and correctness issues associated with the building of such circuits.

We propose that many of the problems of the design of large circuits can be solved by using a formal design notation based upon the functional programming paradigm, that embodies design concepts that have been used extensively as the framework for software construction. The design notation should permit parallel, sequential, and recursive decompositions of a design into smaller components, and it should allow large circuits to be constructed from simpler circuits that can be embedded in a design in a modular fashion. Consistency checking should be provided as early as possible in a design. Such a methodology would structure the design of a circuit in much the same way that procedures, classes, and control structures may be used to structure large software systems.

However, such a design notation must be supported by tools which automatically check the consistency of the design, if the methodology is to be practical. In principle, the methodology should impose constraints upon circuit design to reduce errors and provide 'correctness by construction'. It should be possible to generate efficient and correct circuits, by providing a route to a large variety of design tools commonly found in design systems: simulators, automatic placement and routing tools, module generators, schematic capture tools, and formal verification and synthesis tools.

Acknowledgements

Scientific Research in the area of Microelectronic Circuit Design has traditionally never been performed in Ivory Towers. The work reported on in this thesis was partly done in collaboration with the other members of the VLSI Design Group at the University of Newcastle upon Tyne.

Design of the STRICT language was done in collaboration with Martin McLauchlan, and involved extensive discussions with Roy Campbell, David Kinniment and Harry Whitfield. The STRICT grammar was written in collaboration with Martin McLauchlan.

David Kinniment, as the most experienced member of the group, was heavily involved in discussions about all aspects of the language and design system. He is the author of the GAELIC layout subsystem.

SERC sponsored Research Associates and Research Students programmed most of the subsystems discussed in the thesis. They include Frank Burns, Caroline Wawman, Adrian Robson, Chris Parkin, Mike Fletcher, and Tim Busfield.

I would like to thank all of my colleagues for creating such a positive working atmosphere, particularly since this involves collaboration between groups in different departments (in fact, in different faculties). Special thanks go to David Kinniment, Harry Whitfield and Fred de Geus for their efforts in reading and commenting upon drafts of this thesis. SERC is gratefully acknowledged for its financial support for the work at Newcastle.

I would finally like to thank my wife Lydia and my two daughters Felicity and Helen for all their love and support over the years.

LIST OF PUBLICATIONS

The following papers were published as a result of the work reported upon in this thesis.

Journal/Conference publications

Campbell, R.H., Koelmans, A.M., McLauchlan, M.R., "STRICT: a design language for strongly typed recursive integrated circuits", IEE Proc., Vol. 132, Pts E and I, no. 2, March/April 1985, pp. 85–96.

Koelmans, A.M., McLauchlan, M.R., Robson, A.P., "The STRICT language and Design Methodology", Proc. 1987 Electronic Design Automation Conf., London, July 1987, pp. 79–86.

Koelmans, A.M., McLauchlan, M.R. and Kinniment, D.J., "Asynchronous extensions to the STRICT high level design system", Proc. Int. Conf. on Custom Microelectronics, London, 1988, pp. 57–62.

Kuszynski, C.A., Busfield, T., Koelmans, A.M., McLauchlan, M.R., Kinniment, D.J., "Graphical Representation of a Hardware Description Language", IEE Proc., Vol. 137, Pt. E, No. 6, Nov. 1990, pp. 462–468.

Burns, F.P., Kinniment, D.J., Koelmans, A.M., "Correct interactive transformational design of DSP hardware", Proc. EDAC–91, Amsterdam, 1991, IEEE Press, pp. 16–23.

Burns, F.P., Kinniment, D.J., Koelmans, A.M., "STRIDE: a tool for formal interactive system synthesis", IEE Proc. Comp. Dig. Tech., Vol. 141, No. 6, Nov. 1994, pp. 347–355.

Book Chapters

Koelmans, A.M., Burns, F.P., Kinniment, D.J., "Use of a theorem prover for transformational synthesis", In: *Algorithmic and Knowledge Based CAD for VLSI*, Taylor and Russell (Eds), Peregrinus (IEE Press), 1992, pp. 24–46.

Other publications

Campbell, R.H., Koelmans, A.M., McLauchlan, M.R., "STRICT: a design language for strongly typed recursive integrated circuits", Computing Laboratory, University of Newcastle upon Tyne, Technical Report TR211.

Koelmans, A.M., McLauchlan, M.R., Robson, A.P., "The STRICT language and Design Methodology", Computing Laboratory, University of Newcastle upon Tyne, Technical Report TR244.

Koelmans, A.M., McLauchlan, M.R. and Kinniment, D.J., "Asynchronous extensions to the STRICT high level design system", Computing Laboratory, University of Newcastle upon Tyne, Technical Report TR272.

Kuszynski, C.A., Busfield, T., Koelmans, A.M., McLauchlan, M.R., Kinniment, D.J., "Graphical Representation of a Hardware Description Language", Computing Laboratory, University of Newcastle upon Tyne, Technical Report TR318.

Burns, F.P., Kinniment, D.J., Koelmans, A.M., "Correct interactive Transformational Synthesis of DSP hardware", Computing Laboratory, University of Newcastle upon Tyne, Technical Report TR321.

Koelmans, A.M., Burns, F.P., Kinniment, D.J., "Use of a theorem prover for Transformational Synthesis", Digest of Abstracts, Colloquium on Formal and Semiformal methods for Digital Design, Computing and Control Division, IEE, London, January 1991.

Koelmans, A.M., Burns, F.P., Kinniment, D.J., "Use of a theorem prover for transformational synthesis", Computing Laboratory, University of Newcastle upon Tyne, Technical Report TR426.

TABLE OF CONTENTS

1. INTRODUCTION	6
1.1. Background	7
1.2. Hardware Description Languages	8
1.3. Examples of HDLs	10
1.4. Design Systems	12
1.4.1. Overview	12
1.4.2. Design path	13
1.5. Functional programming	15
1.5.1. Recursion	15
1.5.2. Functional languages	16
1.6. Thesis overview	17
1.7. Author's contribution	18
2. HARDWARE DESCRIPTION LANGUAGES	20
2.1. Introduction	21
2.2. Requirements for HDLs	21
2.3. ELLA	24
2.4. Verilog	27
2.5. VHDL	28
2.6. UDL/I	31
2.7. Discussion	32
3. LANGUAGE DEFINITION	36
3.1. Introduction	37
3.2. Requirements for STRICT	37
3.3. Design concepts	39
3.4. The Block	40
3.4.1. Basics	40
3.4.2. Design parameters	40
3.4.3. Interface specifications	41
3.4.4. Block declarations	42
3.4.5. Inherit statement	44
3.4.6. Define statement	45
3.4.7. Assert expression	45
3.4.8. Size expression	45
3.5. Type declaration	45
3.5.1. Basics	45
3.5.2. Type representations	46
3.5.3. Representational functions	48
3.6. Behavioural specifications	49
3.6.1. Basics	49
3.6.2. Example	49
3.6.3. States	50
3.6.4. Initial statements	51
3.6.5. Invariants	51
3.6.6. Simulator Timing model	52
3.6.7. Selection and Causal Guards	53

3.6.8. Ambiguity Time Delays	54
3.6.9. Effects	54
3.6.10. Duration time delay	55
3.6.11. Exception conditions	55
3.6.12. Behavioural functions	56
3.7. Structural specification	57
3.7.1. Basics	57
3.7.2. Use of recursion	57
3.7.3. Inherit statement	58
3.7.4. Instancing components	58
3.7.5. Hierarchy flattening	59
3.7.6. Placing the components	59
3.7.7. Module generator interface	60
3.7.8. Interconnecting the components	60
3.7.9. Connecting the interface	61
3.7.10. Control modelling	62
3.8. Design organisation	66
3.9. Examples	67
3.9.1. Library cell	67
3.9.2. Half adder	69
3.9.3. Full adder	71
3.9.4. n-input OR gate	73
3.9.5. n-bit register	76
3.9.6. Systolic Array	79
3.9.7. Sigma function	83
3.9.8. Error Corrector	84
3.9.9. Traffic Light Controller	86
4. DESIGN METHODOLOGY	90
4.1. Introduction	91
4.2. Design Methodology	91
4.3. Design system	92
4.4. Syntax directed editor	95
4.5. Simulator	95
4.6. Layout generator	96
4.7. Module generation	96
4.8. Graphical tools	97
4.9. Formal Verification tools	98
4.10. Transformational synthesis	98
4.11. Interfaces with other systems	99
5. SYSTEM OVERVIEW	100
5.1. Introduction	101
5.2. Author's contribution	101
5.3. The SAGA editor	102
5.4. The BUILDER	104
5.5. Procedural interfaces	104
5.5.1. Parse tree interface	104
5.5.2. Design Hierarchy Tree interface – simulator	105
5.5.3. Design Hierarchy Tree interface – layout	105
5.6. Tool interfaces	105

5.6.1. The Simulator interface	105
5.6.2. The Layout interface	106
5.6.3. The Module Generator interface	106
5.6.4. The Viewer	106
5.6.5. The Boyer–Moore interface	106
5.6.6. The Transformer interface	107
5.7. Code developed by the author	107
5.8. System versions	109
5.9. Some code statistics	109
6. THE FRONT END	111
6.1. Introduction	112
6.2. Setting up the Editor	112
6.3. Overview of Operation	113
6.4. Editor output	113
7. THE BUILDER SYSTEM	114
7.1. Introduction	115
7.2. The 'builder' procedure	115
7.2.1. Requirements	115
7.2.2. Overview of problems	117
7.2.3. Chosen solution	119
7.2.4. Success of method	120
7.3. Layout output	120
7.4. Simulator output	121
7.5. Other subsystems	121
8. INTERFACE TO TRADITIONAL TOOLS	122
8.1. Introduction	123
8.2. Simulator	123
8.2.1. Extraction	123
8.2.2. Language interface	123
8.2.3. The stack language	124
8.3. Layout interface	128
8.3.1. GAELIC interface	128
8.3.2. EDIF interface	129
8.4. Module generator interface	129
8.4.1. PLA generation	130
9. THE VIEWER	131
9.1. Introduction	132
9.2. Author's contribution	133
9.3. Extraction	133
9.4. The example	133
9.5. Traversing the hierarchy	135
9.6. Structure and interconnect	136
9.7. Subcells	139
9.7.1. Windowing	139
9.7.2. Instance information	139
9.7.3. Port information	140
9.7.4. Net information	140

9.8. Dual text—graphics representation	141
9.9. Control synthesis	143
9.10. Resource allocation	147
10. BOYER—MOORE INTERFACE	150
10.1. Introduction — formal methods	151
10.2. Boyer—Moore versus HOL	151
10.3. The Boyer—Moore prover	152
10.4. A proof example in Boyer—Moore	154
10.5. Mathematical proof of hardware specifications	157
10.6. Extraction	158
10.7. Output	158
10.8. Results	160
11. THE TRANSFORMER	163
11.1. Introduction	164
11.2. Author’s contribution	165
11.3. Transformational synthesis	165
11.4. Basic ideas	165
11.5. Extraction	166
11.6. Design procedure	166
11.7. Overview of operation	167
11.8. Results	181
12. CONCLUSIONS	182
12.1. Introduction	183
12.2. Advantages	183
12.3. Disadvantages	187
12.4. Practical experience	191
12.5. Final conclusions and future work	192
13. REFERENCES	193
A. SAMPLE OUTPUTS	210
A.1. Simulator	211
A.2. Layout	213
A.3 PLA generator	216
A.4. EDIF output	220
A.5. Boyer—Moore Prover output	231
A.6. Transformer output	239
B. STRICT SYNTAX	244

LIST OF FIGURES

Fig. 1. Typical design cycle	14
Fig. 2. Ideal design path	14
Fig. 3. Divider module	63
Fig. 4. Structure of inverter	69
Fig. 5. Structure of half adder	71
Fig. 6. Structure of full adder	73
Fig. 7. Recursive structure of n -input OR gate, $n > 4$	76
Fig. 8. Original hierarchy	78
Fig. 9. Flattened hierarchy	79
Fig. 10. Overview of the STRICT system	103
Fig. 11. Developed code	108
Fig. 12. Top level view	134
Fig. 13. View Structure option	135
Fig. 14. Option $N > 1$ selected	136
Fig. 15. Illustrating Join	137
Fig. 16. Illustrating split, first part	138
Fig. 17. Illustrating split, second part	138
Fig. 18. Illustrating split, combined	138
Fig. 19. A window on an area	140
Fig. 20. Net information	141
Fig. 21. Find text	142
Fig. 22. Divider structure	145
Fig. 23. Divider structure – pin names	145
Fig. 24. Control flow Petri Net	146
Fig. 25. Gantt resource allocation chart	148
Fig. 26. Main screen – control section present	149
Fig. 27. Library operators	167
Fig. 28. Main screen	169
Fig. 29. After click on leftmost multiplication	172
Fig. 30. After deletion of multiplication sub-tree	173
Fig. 31. Showing rewrite rule	174
Fig. 32. Applying rewrite rule	175
Fig. 33. Applying rule 2	176
Fig. 34. Result of rule 2	177
Fig. 35. Applying rule 3	178
Fig. 36. Final result	179
Fig. 37. Hardware allocation	180
Fig. 38. Block diagram of final result	181
Fig. 39. Simulator screen dump	212
Fig. 40. Full adder layout	213
Fig. 41. 6 bit register layout	214
Fig. 42. Systolic array layout	215
Fig. 43. PLA plot	219

1. INTRODUCTION

1.1. Background

The rapid pace of development of integrated circuit manufacturing technology has forced a great deal of change upon the designers of circuits. During the 1960's and 1970's, chip designers used to draw their designs by hand, by entering the appropriate mask shapes using low level graphical tools. Verification usually consisted of visual inspection of the mask patterns. Such techniques are simply no longer feasible now that the number of gates on a chip has increased so dramatically.

The idea of using languages to describe chip designs dates back to the 1950's. However, only during the 1970's did computers become powerful and user friendly enough to allow development of computer languages for circuit layout to start in earnest. The first conference entirely devoted to Hardware Description Languages was held in 1973 at Rutgers University; it is currently a two-yearly event with a still rapidly growing attendance.

A typical example of a language developed during the 1970's is CIF [51], which describes layouts in terms of polygons (i.e. mask patterns), and in which it is the responsibility of the designer to enforce the design rules of the technology at hand. Because this design method was just as error prone as layout by hand, extensive simulation and verification was necessary. Despite the fact that most languages like CIF have macro facilities, a description of a non-trivial circuit is extremely verbose and hence unreadable. In order to provide these low level languages with more power of expression, layout systems were developed consisting of a set of procedures embedded in a conventional high-level language, each procedure generating a certain low level description. One of the earliest examples of these systems was LAP [46]. Despite the fact that systems like LAP make the familiar constructs of programming languages, like loops and expressions, available, a circuit description is still likely to be verbose because the designer is mainly concerned with gate placement.

The main difficulty with languages such as CIF is the large conceptual gap between the specification of the design on the one hand, and its implementation on the other hand. *Specification* defines the intended behaviour or functionality of the chip, whilst *implementation* refers to its physical properties – the actual layout, for example. It is obvious that

these are very different. The problem for the designer is that he somehow needs to translate the specification into an implementation, whilst hoping that no errors occur during the mental processes that direct the translation.

Whilst a low level design method was acceptable for small chip designs, it rapidly became obvious that more advanced methods were necessary to design much larger chips, without losing all confidence in the correctness of the designs. This led to a rapid increase in research into CAD tools for VLSI design, and an equal interest in the development of Hardware Description Languages, or HDLs for short.

This thesis is concerned with the design of an HDL called STRICT (meaning Strongly Typed Recursive Integrated Circuits), and an investigation into its suitability as an input language for use with a number of important CAD tools.

1.2. Hardware Description Languages

Languages that describe integrated circuits can do this in fundamentally different ways.

First there is the *structural* view of a design. This describes the design in terms of interconnected hardware components. The designer who writes structural descriptions has already decided how he is going to implement his algorithm. Structural descriptions can take place at several levels: *geometrical*, in which the design is described in terms of mask shapes, *electrical*, in which the individual transistors form the basic units, *switch*, in which transistors are reduced to simple switching devices, *logic*, at which boolean functionality is the basic operation, *register transfer*, in which the design is regarded as a collection of registers between which data passes, and *architectural*, which regards the design as a processor model with large building blocks. The geometrical view is the lowest level (SPICE [64] being the most widely used tool for modelling at this level), and the architectural the highest. Clearly, the higher the level, the more work needs to be done to translate this down to the lowest level; however, if the translation can be done automatically, much higher productivity can be achieved, and there will be considerably less chance that errors will be introduced.

The second view is the *behavioural* view. In this view, the algorithm which the circuit is to perform is described. No description is provided as to how this can be achieved in terms of hardware components. Since any designer must always start off with some sort of specification, the behavioural view is always present during the design process. However, the unambiguous specification of a complex algorithm is a difficult and time consuming task, perhaps requiring the use of formal mathematical techniques. This would only be beneficial if appropriate design tools were available to translate the specification into an efficient structure. Behavioural descriptions also come at different levels, e.g. description of current flow behaviour at the lowest level, state transitions and (possibly conditional) signal and data flows at higher levels, and mathematical equations at the highest level. All levels require the inclusion of *timing* information – a description of the time bounds within which the operations must take place.

Behavioural descriptions are commonly referred to as high level descriptions, while structural descriptions are regarded as low level descriptions. The process of translating a high level behavioural description into an efficient structure through the use of appropriate design tools is called *high level synthesis* [49].

The benefits of using high level behavioural HDLs can be summarised as follows. The designer can achieve much greater productivity, since high level descriptions are much more concise. The designer thinks at the conceptual level, instead of at the gate level, which should be more natural and therefore less error prone. High level behavioural languages should allow improved design quality, through the use of automatic optimisation techniques and the possibility of rapidly exploring a number of design alternatives (and choosing the "best" one). They allow consistency between different levels of abstraction, through what is commonly called 'correctness by construction'. They allow people who are not necessarily experts on manufacturing details and design rules (so called "silicon hackers") to become competent designers. Designs in high level languages tend to be technology independent, so designs are not specifically targeted towards a specific technology or cell library. Modules, once debugged and verified, can be reused more easily in other designs.

Finally, if they are written in a suitable formalism, high level descriptions allow *formal verification*.

Given all this, it is no surprise that recent research into HDLs has tended to move towards behavioural languages. In many cases, researchers have investigated the suitability of already existing formalisms. In practice, many newly developed languages combine descriptions of both behaviour and structure to try to obtain the best of both worlds.

1.3. Examples of HDLs

There are almost as many HDLs as there are programming languages. We mention a few typical ones, and then concentrate on what we feel are currently the most important HDLs, namely VHDL, Verilog, ELLA, and UDL/I.

An example of a structurally oriented language is MODEL [34]. MODEL makes it possible to build systems in a modular, top down way. The language has programming language constructs such as loops and conditionals, and it makes parameterisation of modules possible. Connections between modules are indicated explicitly by arrows (the equivalent of the assignment statement). The target architecture is the gate array.

There are many other languages in the same class as MODEL. Since they all describe netlists in one form or another, they can all be regarded as semantically equivalent.

ISPS [4], is a register transfer language which is targeted towards the specification of processor architectures. The language therefore has high level constructs specifically for this purpose, and allows efficient translation to silicon, but it cannot easily be used for the design of general hardware.

Another one of the older generation languages with behavioural features is MacPitts [63]. This is an abstract, Lisp-like language which makes it possible to specify circuits by (possibly parallel) algorithms which are converted by the compiler into a CIF description. Because the compiler enforces design rules, correctness by construction is achieved. One disadvantage of the MacPitts approach is the fact that, like ISPS, the target architecture is embedded in the semantics of the language, which limits the range of its applications.

At around the time that behavioural HDLs became the focus of intensive research, the debate about imperative versus functional programming was also going on in the software world, after Backus' Turing Lecture [3] and earlier research into software verification. *Imperative* programming languages are ones like FORTRAN and PASCAL. Imperative programs consist of sequences of statements that must be executed sequentially; they make use of global variables, and subroutines are allowed to modify those variables. *Functional* programs consist of a collection of statements that are performed in response to external stimuli; no sequence is assumed, and side-effects on variables are not allowed. The functional programming style is also frequently referred to as *declarative*. As the advocates of functional programming point out, languages based on the imperative programming style suffer from correctness problems. Specifications in such languages are hard to verify, because they cannot be easily characterised in a mathematically rigorous manner. Their use of variables and functions with side-effects allows very complex and error prone specifications to be written. In functional programs, by contrast, the use of recursion to achieve iteration, and the absence in functions of side-effects on variables, allows the application of mathematical induction and other mathematically based techniques to achieve proof of correctness.

Because of the popularity of imperative programming languages, the first truly general high level behavioural HDLs were inspired by them. Dacapo[59] and Silage[69] are good examples. There were many others. It rapidly became clear, however, that such languages would not be a suitable vehicle for the formal specification and verification of hardware designs.

As a result, many researchers started investigating non-imperative languages for use in hardware design, including already existing programming languages. These include formalisms such as denotational hardware models [29], a variation on the FP functional programming language, called muFP [62], temporal logic [53], SCCS [13], Algebraic approaches [30], Concurrent Prolog [65], Higher Order Logic [31], and type theory [37]. Many of these formalisms allow pure specifications to be written, without any reference to possible hardware implementations. The problem with this approach is that very sophisti-

cated software systems are required to translate such specifications into silicon *in an efficient manner*. In many cases it is not possible for the designer to have any influence at all on the way that the hardware is generated.

During the second half of the 1980's three popular HDLs emerged: Verilog, VHDL and ELLA. ELLA (ELectronic design LAnguage[52]) was a project initiated by the UK DoD, and developed by a team at RSRE. VHDL (VHSIC Hardware Design Language[41]) was a large effort on behalf of the US DoD, and is now an IEEE standard. ELLA was widely used within the UK, and the language was used to specify and design a number of substantial chips. Verilog[72] was first developed as a proprietary simulation product by a company called Gateway, which later merged with Cadence, and is widely used. A more recent development is the language UDL/I, as a result of a Japanese standardisation effort.

We will take a look at the features and deficiencies of these languages in chapter 2.

1.4. Design Systems

1.4.1. Overview

Design systems usually consist of a set of software tools to enter a design in some form, validate it, and translate it into silicon. A good VLSI design system provides descriptions which are consistent at all levels. It must be easy to use, allow convenient generation of test vectors and test hardware, and also address issues such as performance of the design (speed, power, function), the size of chip (and therefore, its cost), design time, and others. A variety of design tools is essential, to tackle the different areas that require attention. This will involve making trade-offs between parameters. Most modern design systems use HDL descriptions as an input medium. In recent years, input from VHDL or Verilog has become essential. EDIF [24] is widely used for interchange between different CAD systems.

Typically, a design system comprises tools for

- layout (which includes floor planning, placement, routing, design rule checking, etc.);
- simulation (at various levels);

- schematic capture, i.e. logic entry using graphical tools;
- timing verification;
- module generation, particularly for PLA and RAM;
- language parsers, to allow input from HDLs (particularly VHDL)
- formal verification/synthesis tools, if the underlying formalisms will allow it.

Most high level design systems originate in the academic research environment. SAGE [22] (Edinburgh), Lambda [27] (Brunel) and Veritas [35] (Kent) are systems developed in the UK. Other notable systems are Cathedral [69] (IMEC), CMU-DA [68] (Carnegie-Mellon), OCCAM to Silicon [47] (Meiko), DSL[16] and YSC [10] (IBM). Notable commercial systems include those by CADENCE, Viewlogic and Mentor Graphics. Because of the importance of VHDL, many research efforts in the area of VHDL based synthesis are reported, e.g. [18, 23].

1.4.2. Design path

A typical current design path is shown in Figure 1. This is a typical structural approach to designing integrated circuits. After the initial specification, provided by the customer, the designer first makes decisions about the overall architecture of the chip. The design effort then moves gradually down to the silicon level using appropriate tools to perform translation, simulation and design of new modules. This approach requires that a number of separate descriptions be verified for equivalence, and is very labour intensive and error prone.

The behavioural approach, by contrast, would be to synthesize the design automatically after the top level specification was completed. One would then get the design path shown in Figure 2. The designer specifies the problem in terms of the algorithm to be implemented using an appropriate HDL, and leaves the design system to do the rest.

In practice, the design space may be too large for such an approach, and some structural design will have to be done by the designer, perhaps using appropriate high level tools that preserve correctness.

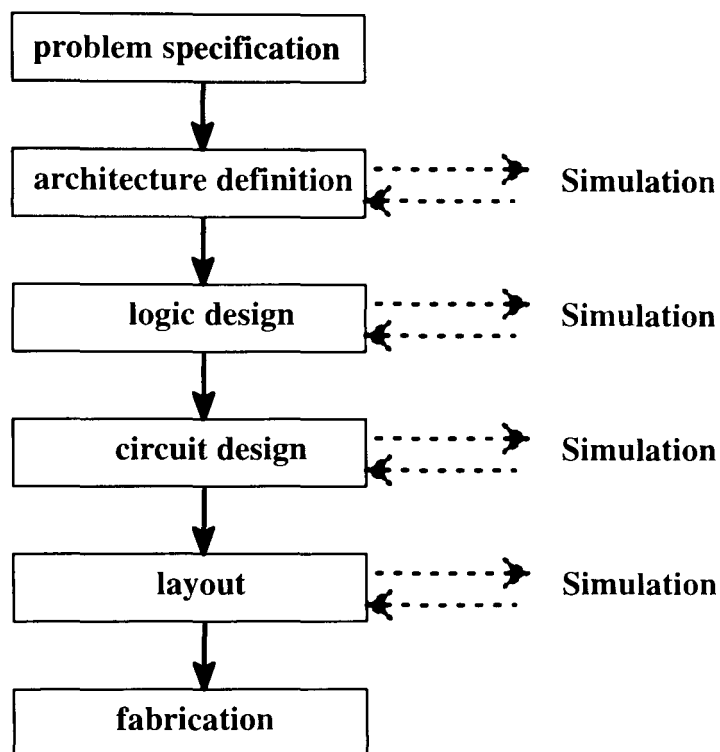


Fig. 1. Typical design cycle

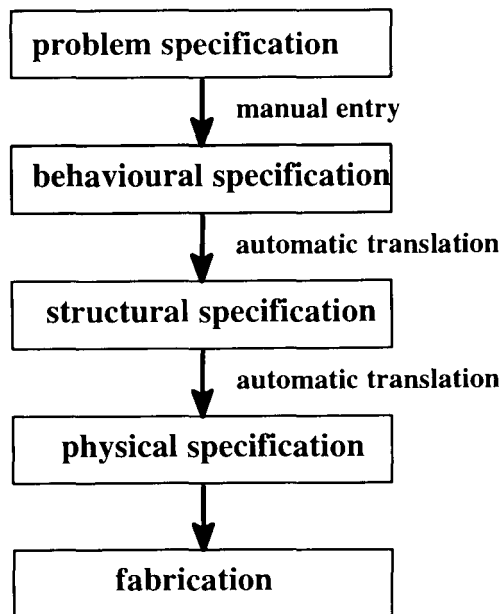


Fig. 2. Ideal design path

1.5. Functional programming

Functional programming is a discipline that has been around since the early 1960's, and it was put firmly on the map by Backus [3] in his Turing Award lecture. Functional programming has been advocated as the solution to the so called 'software crisis' – the fact that it becomes more and more difficult to control the complexity and cost of software written in traditional imperative programming languages such as Pascal or C. Functional programming is also advocated strongly as a convenient vehicle for VLSI applications, because of the inherent parallel nature of both functional programs and VLSI chips.

Imperative languages tend to become very complex and error prone as their size increases. This is due to a number of basic features in such languages:

- The possibility of performing arbitrary jumps between code segments;
- the large variety of language constructs to perform repetition;
- limitations in what procedures and functions can return;
- global variables, and the possibility of assigning values to them from within functions and procedures (so called *side-effects*).

It is the possibility of side-effects that is the most serious deficiency in imperative programming languages. It allows the same expression, evaluated twice in succession, to produce different results. This makes it extremely difficult to subject such code to mathematical proof techniques.

Functional programs therefore ban side-effects. They consist of collections of functions. There are no global variables, and therefore no side-effects. A function will always return the same result if called with the same parameters. The most important data structure is the *list*, which may have an unlimited length. Functions use *recursion* to achieve iteration. This makes them amenable to the mathematical proof technique called *induction*. This in turn makes it far easier to verify the correctness of functional programs.

1.5.1. Recursion

Recursion is often regarded with suspicion by novice programmers. A recursive description is much more like a specification than an implementation – in other words, it describes

what needs to be done instead of how it is to be done. The classic example of recursion is of course the Towers of Hanoi problem, which can be expressed in Pascal as follows:

```
procedure hanoi(n: integer; first, last, temp: char);
begin if n > 0 then begin
    hanoi(n-1, first, last, temp);
    writeln('move disk from ', first, 'to ', last);
    hanoi(n-1, temp, first, last)
end
end
```

This definition is very short. Imperative versions are much longer and more difficult to understand, once the basic concept of recursion has been understood. The parameter 'n' is used to detect the termination of the recursion; otherwise, no new values are assigned to variables. The description could be subjected to formal treatment if required.

Another example, to show the use of recursion to perform a useful calculation, is the base-2 logarithm:

```
function log2(n: integer): integer;
begin if n = 1 then log2 = 0
      else log2 = 1 + log2(n div 2)
end
```

Any form of repetition can be expressed in recursive form, so a programmer will not be subjected to unnecessary restrictions if he limits himself to its use.

1.5.2. Functional languages

There are several functional programming languages available. One of the oldest ones is LISP [80]. In LISP, the 'log2' function looks as follows:

```
(define ((log2 (lambda (n)
  (cond ((equal n 1) 0)
        (T (plus 1 (log2 (div n 2))))
  ))))
```

The LISP notation has a number of advantages. It is very general, easy to parse, concise, powerful, and extensible. It has been used to define other languages, such as EDIF. However, not everyone likes it. Design engineers do not like the abundance of brackets and the absence of keywords, for example. Other languages simply look much more understandable.

In KRC [79], the 'log2' function looks as follows.

$$\text{log2 } 1 = 0$$

$$\text{log2 } n = 1 + \text{log2 } (n / 2)$$

KRC obviously looks concise and readable, and would be a highly suitable basis for a language to describe algorithms to be implemented in silicon. However, a notation for designing VLSI circuits would have to be much more than just a set of equations in order to be acceptable to designers. It would have to have suitable keywords, and allow description of hierarchy and timing information amongst other things. The KRC notation is however an inspiration for the functional parts of an HDL.

1.6. Thesis overview

This thesis describes a new HDL called STRICT, which is inspired by functional programming languages such as KRC. As mentioned earlier, the designers of functional languages argue, we believe, with some force, that programs written in such languages are amenable to rigorous mathematical proof techniques. It was therefore investigated whether they were suitable for application in the VLSI design area, where correctness of the design is absolutely crucial.

In this thesis we adopt the following lexicographical conventions. Italics are used to emphasize important words. Boldface is used for keywords from various languages, whenever they are surrounded by normal text. Whenever we show self-contained examples of the STRICT language, we normally use uppercase letters to indicate keywords. These examples are shown in Courier font. Single quotes are used to indicate identifiers and syntactical constructs. We use uppercase letters for names of languages and products (e.g.

STRICT, EDIF, CADENCE etc.). Finally, when we use the word 'he' we mean 'he or she', and when we use 'his' we mean 'his or her'.

The thesis is structured as follows. In chapter 2, we present and discuss the most important HDLs currently available, and define the requirements for the STRICT language. In chapter 3 we define the STRICT language and present basic examples of hardware modules defined in STRICT. In chapter 4, we explore the practical use of the language in the design process. In chapter 5, we present a brief overview of the design system. The individual components of the design system are then described in subsequent chapters: the editor in chapter 6, the builder module in chapter 7, the interface to 'traditional' design tools such as the layout system, the simulator and the PLA generator in chapter 8, the viewer in chapter 9, the theorem prover interface in chapter 10, and the transformer interface in chapter 11. The thesis then finishes with conclusions and references, followed by two appendices, the first one with some example output from the various parts of the system, and the second one with the STRICT syntax in the form of railroad diagrams.

1.7. Author's contribution

The work reported on in this thesis was done in collaboration with the other members of the VLSI Design Group at the University of Newcastle upon Tyne.

Design of the STRICT language was principally done in collaboration with Martin McLauchlan, and involved extensive discussions with other members of the group. The STRICT grammar was written and debugged in collaboration with Martin McLauchlan.

Part of the work involved minor modifications to the SAGA software, principally the lexical analysis routines.

The major part of the work involved writing the builder module and the various interfaces to the subsystems, as well as developing the ideas behind these interfaces. This was all done by the author, and amounts to nearly 20,000 lines of Pascal code. This code is described in much more detail in chapter 5.

The author wrote the EDIF generator and the Boyer-Moore input generator subsystems.

The author was responsible for the development of the ideas that led to the development of the viewer and the transformer subsystems. However, SERC sponsored Research Associates and Research Students actually programmed the simulator, layout, PLA generator, viewer and transformer subsystems.

2. HARDWARE DESCRIPTION LANGUAGES

2.1. Introduction

In this chapter we first look at the most essential requirements for HDLs, and then discuss the four most important HDLs currently available: ELLA, Verilog, VHDL and UDL/I. In each case, we illustrate the language by presenting an example.

2.2. Requirements for HDLs

It is possible to identify some important features that all HDLs should have in order to be of interest to designers. Most of these features have been discussed briefly in the introduction, but are discussed in greater detail here. Aylor et al [73] present a thorough, but otherwise typical mid 1980's view, which excludes such important issues as formal verification and (formal) high level synthesis. These issues have become of much more concern during the late 1980's and 1990's. We specifically exclude analog design features, which have recently been added to VHDL, from the discussion.

- It is absolutely essential that a language should be able to describe both behaviour and structure. Behaviour and structure should be described separately, since they refer to completely different concepts, so it is necessary to make a distinction between the two. Behaviour describes a design module as a *black box* that produces outputs as a result of inputs and internal states. Structural description describes the internal decomposition of a module (which leads to the introduction of a design hierarchy), and also includes the interface of a design module to its environment.
- A language should allow description of the timing properties of a design. Description of timing properties allows accurate (usually event driven) simulation of a design. Timing description should allow for such features as timing granularity, propagation delay, and clock properties. Description of timing properties is part of a behavioural description.
- Hierarchy should be an essential feature of any language. It allows a complex problem to be partitioned into smaller, less complex ones. A language should therefore be capable of describing designs at different levels, both in the behavioural and structural domain. It should be possible to describe collections of gates as well as entire

systems at the algorithmic level. In the behavioural domain, it should be possible to describe these levels efficiently, for example as a set of logic equations at a low level, a finite state machine at a higher level, or a program (possibly with explicit concurrency and synchronisation information) at the highest level. There should be consistency between adjacent levels, allowing design tools to achieve automatic translation and correctness by construction.

- Each level should allow for simulation, even if the level below is incomplete. This allows the designer to gain confidence in the correctness of the current level before investing time and effort in an incorrect design that will have to be discarded later. An important issue here is the availability of pre-defined high level primitives, for example ones that relate to data communication and timing. Such primitives allow for much more efficient simulation than designer specified ones. There is therefore a trade-off between efficiency and the number of pre-defined primitives.
- A language should be used in conjunction with tools that can work at all levels, and should therefore allow transfer of design data to/from these tools at all different levels. This should also improve the effort of data management of large designs, possibly in a distributed fashion. Ideally, a language should be independent of its design tools, that is, it should not require the use of special constructs that are there to interface to a specific (perhaps proprietary) tool, but are of no use to other tools. If for some reason such constructs are required, they should be optional.
- A language should have an uncluttered and unambiguous syntax. Verbosity makes it more difficult and time consuming to understand an existing design, and it increases the likelihood of making mistakes when entering the design.
- A language should have unambiguous semantics. This prevents different interpretations of the same descriptions, and allows re-use of developed designs, possibly by different designers working in completely different environments. It also clearly aids the formal development of tools, e.g. through the use of compiler generators etc.

- A language should support abstract data types. The use of abstract data types allows concise and readable descriptions (which are in effect a form of documentation), and allows a designer to extend the language for his own purpose, perhaps to prepare for easy migration to new design tools in the future. Used in conjunction with strong typing, particularly with regards to the input and output ports of a module, abstract data types should assist consistency checking within a design, and therefore design correctness.
- New chip designs are hardly ever designed from scratch. In many cases, it will be possible (and indeed desirable) to use components designed and verified earlier on. A language should therefore allow existing components to be re-used. This can be achieved by means of modularity, that is, the ability to split designs into modules that perform a well defined and self contained function. Re-use of components leads to higher design quality if a number of alternatives are available (typically for low level modules such as adders or multipliers, which can be implemented in several different ways), and the designer has the option of choosing the best possible alternative for his particular design.
- A language should allow the design of generic components, i.e. families of designs which are available through a single, parameterised description. This will clearly reduce design effort, and also reduce the chance of design bugs.
- A language should allow some form of formal verification. This is an essential requirement in today's very complex systems, where exhaustive simulation is no longer possible due to the extreme complexity of the designs. In formal verification, mathematical formalisms and techniques are used to check or verify certain properties of a design, using a tool such as a theorem prover. This will be most difficult at the higher levels of a design. Lower level proofs can usually be done by exhaustive proof of simple cases, because of the lower complexity of modules at the lower levels of a design.

- A language should allow high level synthesis, that is, the automatic conversion of a behavioural description into a structure. Conversion of low level logic equations or structures is regarded as low level synthesis, and is currently widely available in design systems. This requires sophisticated tools to be available. High level synthesis greatly increases productivity and helps to achieve design correctness, and is now widely used in industrial design environments. In formal synthesis, a behavioural description is converted into a structure. This is usually not a problem at lower design levels, but may be difficult at higher design levels where abstractions such as concurrent processes and their synchronisation may be available.

We investigate the four languages mentioned above in terms of these features, by giving examples of descriptions. We also investigate their facilities for writing behavioural descriptions and structures in a recursive manner.

2.3. ELLA

The aims of ELLA were defined as follows[52]: To provide a design aid that could be used at all levels of abstraction; to allow the designer the freedom of setting his own design style/methodology; and to allow easy mappings between different levels of abstraction. The basic philosophy is to offer very few built-in primitives and data and signal types, but rather to provide a vehicle for the designer to build his own models from the bottom up.

ELLA has strong typing, and has the facility to build abstract data types through the use of type declarations. For example,

```
TYPE bool = NEW (h|l),
      int  = NEW i/(0..63),
      lint = NEW li/(0..3969),
      llint = NEW lli/(0..500000)
```

This declares a new type 'bool' which can take the values 'h' and 'l', and three integer subrange types with the bounds shown in brackets. ELLA does not have pre-defined types, so the typing mechanisms is a very important part of the language. ELLA supports hier-

archy through the use of modules, which are called 'functions' and are started by the keyword **fn**, followed by input/output declarations. An example of a simple function with two boolean inputs and one boolean output is

```
FN BLOCK = (bool: in1 in2) -> bool:
"body"
```

A complete description in the language consists of a set of *nodes*, each of which has an associated behaviour, with signals flowing between the nodes by interconnecting them appropriately. Because nodes may declare and use sub-nodes, a hierarchy is created. Operation of signal flow is implicitly concurrent.

Although the language is said to be behavioural, a special keyword is required to start an arithmetical expression, and in fact behaviour may be completely absent from a function. An interesting example is the Sigma function (which performs the mathematical summation operation), which in ELLA would be described as follows. Line numbers have been added for the sake of discussion, but are not actually part of the ELLA description.

```
1/ FN ADD = (lint: i1,llint: i2) -> llint: ARITH i1+i2
2/ MAC SIGMA{INT n} = ([n]lint: ip) -> llint:
3/     IF n=1 THEN ip[1] ADD lli/0
4/     ELSE ip[n] ADD SIGMA{n-1}ip[1..(n-1)]
5/     FI.
```

The 'add' function (line 1) adds two integers as expected. The 'Sigma' function takes an n-bit integer as its input (line 2). If 'n' equals 1, the value of the single bit is used (line 3), otherwise the value of bit n is recursively added to the Sigma of the remaining (n-1) bits.

This is a description which is hardly easily readable, and has a rather complex syntax.

A typical ELLA version of the full adder could be as follows:

```
1/ FN FULL_ADDER = (bool: x y cin) -> [2]bool:
2/ BEGIN MAKE XOR: x1 x2,
3/           AND: a1 a2,
4/           OR: or1.
5/ JOIN (x,y) -> x1, (x1, cin) -> x2,
```

```

6/      (x1,cin) -> a1, (x,y) -> a2,
7/      (a1,a2) -> or1.
8/ OUTPUT (or1, x2)
9/ END.

```

This is clearly a structural description. The description specifies two one-bit inputs and one two-bit output (line 1). Lines 2–4 instance a number of gates, which are then interconnected by the 'join' statement of lines 5–7. The 'output' statement (line 8) connects the internal circuit to the outside world.

Another version of the adder could be:

```

1/ FN FULL_ADDER = (bool: x y cin) -> [2]bool:
2/ BEGIN LET xor = XOR(x,y) .
3/ OUTPUT ( (xANDy) OR (cinANDxor), xorXORcin)
4/ END.

```

which could be interpreted both as a behaviour (a set of logic equations) or as structure (if one regards the logic operators as hardware modules performing the appropriate operation, in which case a netlist can be generated).

There are other ways of specifying behaviour in ELLA, for example as a truth table. This requires the use of a 'case' statement, or, in simpler cases, an 'if-then-else' statement.

The ELLA timing model allows for propagation delay, inertial delay and ambiguity delay. This is achieved by means of a standard function called DELAY which can be inserted into functions.

ELLA comes with a set of tools: a language compiler, a multi-level simulator, and a tool called EASE (ELLA Application Support Environment), which includes database support. Anecdotal evidence would support the view that these tools are easy to use and fairly efficient, although the syntax of stimulus files for the simulator is very low level and repetitive. We are not aware of tools that allow the full language to be formally verified.

ELLA allows the description of generic components.

2.4. Verilog

Verilog was first introduced in the mid-eighties as part of a commercial product called Verilog-XL, a simulator package [74]. It is widely used in industry, particularly in the US.

A typical full adder description in Verilog would look like this:

```
1/ module fullAdder(cOut, sum, aIn, bIn, cIn);
2/     output cOut, sum;
3/     input aIn, bIn, cIn;

4/     wire x2;

5/     nand (x2, aIn, bIn),
6/         (cOut, x2, x8);
7/     xnor (x9, x5, x6);
8/     nor  (x5, x1, x3),
9/         (x1, aIn, bIn);
10/    or   (x8, x1, x7);
11/    not  (sum, x9),
12/        (x3, x2),
13/        (x6, x4),
14/        (x4, cIn),
15/        (x7, x6);

16/    assign #5 sum = aIn ^ bIn ^ cIn,
17/           cOut = ( aIn & bIn) | (bIn & cIn) | (aIn & cIn);
18/ endmodule
```

Lines 1–3 declare the input and output ports to the module. Line 4 declares an internal signal. Lines 5–15 describes the internal structure of the module, by using various (pre-defined) components, and connecting them up via internal signals that are used on the fly. Finally, lines 16–17 declare the behaviour, which is a simple logic equation that also specifies a delay of 5 time units.

As the above description clearly shows, Verilog has a large number of built-in models for the most basic logic gates, and a large set of behavioural operators corresponding to the operation of these logic gates. This allows very efficient simulation. The language also has a 'table' statement, allowing the output of a design to be expressed as a truth table.

Obviously, all the essential elements of the description are put together within the module.

In general, behavioural descriptions in Verilog closely resemble statements in the Pascal programming language [72]. A behaviour is usually described in terms of variables (which in turn often correspond to bit patterns on ports), arithmetical expressions on these variables, 'for', 'while', and 'if-then-else' statements, together with appropriate timing conditions, often synchronised to rising or falling clock edges. The language has a 'wait' statement similar to that in VHDL, to allow more general event conditions. The most general case of a behaviour is an endless loop. The language also has an 'initial' statement that allows it to initialise input waveforms and variables before the start of simulation.

Verilog has a limited capacity for explicitly describing concurrency, by enclosing the appropriate statements within the keywords **fork** and **join**.

Apart from simulators, there are also commercially available high level synthesis tools available for this language. We are not aware of the existence of any formal verification tools.

Verilog does not appear to allow the description of generic components.

2.5. VHDL

VHDL was defined, from the mid 1980's onwards, by a large committee on behalf of the American Department of Defense (DoD), in an attempt to introduce some uniformity into the hardware design process, for which the American Defense Industry is a very large application area. Because of this, VHDL has become a widely used language. It is based upon ADA, an imperative programming language which was earlier adopted by DoD as the preferred programming language. The main aim of VHDL is to provide very accurate modelling of hardware systems at all levels.

VHDL supports hierarchy through the use of 'entity' and 'architecture' declarations. It allows data abstraction, since it is based upon the ADA programming language. VHDL allows descriptions both in the structural and behavioural domains, although these cannot be integrated. The language also allows data flow descriptions, which are descriptions of RTL statements which are concurrently executed. Behavioural descriptions can make use of familiar imperative programming language constructs such as 'for' and 'while' loops. The timing model is quite powerful, and the whole language is geared towards event driven simulation.

A VHDL description of a full adder might be as follows:

```
1/ entity FULL_ADDER is
2/ port (X, Y:in BIT;
3/ CIN: in BIT:='0';
4/ COUT, SUM: out BIT);
5/ end FULL_ADDER;
```

This describes the ports of the adder (input ports on lines 2 and 3, output port on line 4).

We might describe the structure as follows:

```
6/ architecture MY_STRUCTURE of FULL_ADDER is
7/ component andgate port(A,B: in BIT; C:out BIT);
8/ end component;
9/ component xorgate port (A,B:in BIT; C:out BIT);
10/ end component;
11/ component orgate port (A,B: in BIT; C:out BIT);
12/ end component;
13/ signal S1, S2, S3: BIT;
14/ begin
15/ X1: xorgate port map(X,Y,S1);
16/ X2: xorgate port map(S1,CIN,SUM);
17/ A1: andgate port map(X,Y,S3);
18/ OR1: orgate port map(S2,S3,COUT);
19/ end MY_STRUCTURE;
```

This declares the required sub-components on lines 7–12. Note that the entire parameter list of each component must be declared, which greatly increases the length of the description and decreases the readability. Internal signals are declared on line 13. Lines 15–18 then describe the connectivity, including another verbose description of the port connections.

Another description provides us with a possible description of the behaviour:

```
architecture BEHAVIOUR of FULL_ADDER is
20/ signal S: BIT;
21/ begin
22/  S <= X XOR Y after 10 ns;
23/  SUM <= S XOR CIN after 10 ns;
24/  COUT <= (X AND Y) OR (S AND CIN) after 20 ns;
25/ end BEHAVIOUR;
```

Lines 22–24 describe the assignments, including delays, that lead to assignment of values to the output ports of the module. This again includes the use of an internal signal. VHDL has many keywords for specifying the length of delays. There are two kinds of delay: inertial and transport. Transport delays can supersede (i.e. cancel) events that have already been scheduled.

The most general form of a behavioural description in VHDL is the 'process' statement, which is in effect an endless loop that contains a program based upon the ADA programming language, enhanced with hardware specific features. This includes a comprehensive range of arithmetical operators and constructs. VHDL allows the declaration and use of (possibly recursive) procedures and functions. Synchronisation is achieved by means of a 'wait' statement, which allows the program to wait for the occurrence of specific conditions on ports, variables, and other processes. Attributes of signals, such as rising edges, are declared by following the name of the signal by an apostrophe, followed by the name of the attribute, optionally followed by a parameter in brackets.

VHDL has comprehensive data management and documentation facilities through the declaration and use of packages and libraries of types and components.

There are formal verification and formal synthesis tools available for VHDL, but these tools normally allow only a subset of the language to be used.

VHDL allows the description of generic components, although this requires the use of special keywords, and the resulting syntax is rather verbose.

2.6. UDL/I

UDL/I [39] is a Japanese HDL that is currently attracting a great deal of attention. It is intended as a standard and is widely used as such in Japan. Among the attractive features of UDL/I are strong support for data management, testing and accurate timing modelling. On the negative side are the wordiness of structural descriptions (in particular netlists). Although the language allows modelling of behaviour, the available constructs appear to be extremely limited, in order to ensure that direct translation into logic is possible. The language appears to be mostly suited for low level structural descriptions and simulation.

An example of a D-type flip-flop in UDL/I is now shown:

```
1/ name      : dff_cl_pr;
2/ purpose   : funcsym, logsyn;
3/ process   : ttl;
4/ inputs    : d;
5/ clock     : clk;
6/ reset     : cl, pr;
7/ outputs   : q;
8/ behaviour_section;
9/ register  : regq delay 1.0ns;
10/  begin
11/    .q := regq;
12/    at rise(.clk) do regq := .d; end_do;
13/    if .cl then reset(regq);
14/        else if .pr then preset(regq); end_if;
15/    end_if;
16/  end;
```



```

17/ end_section;
18/ end;

```

This description includes data management features (lines 1–3), structural features (lines 4–7), and a behavioural description (lines 8–17).

The language obviously supports many hardware oriented constructs. It has a formal definition of the syntax and semantics, in order to ensure that all simulators produce the same results.

UDL/I allows the description of generic components.

2.7. Discussion

Features discussed in this chapter can be summarised briefly in the following table.

Feature	VHDL	ELLA	Verilog	UDL/I
behaviour – structure	yes	mixed	yes	logic level
timing	yes	yes	yes	yes
hierarchy	yes	yes	yes	yes
simulation at each level	yes	yes	yes	yes
tools interfaces	limited	yes	yes	yes
syntax	verbose	difficult	concise	ok
semantics	yes	no	no	yes
abstract data types	yes	yes	no	adequate
re–use and modularity	yes	yes	yes	yes
generic components	yes	yes	no	yes
formal verification	limited	limited	unknown	no
high level synthesis	limited	limited	yes	no
recursive structures	limited	yes	no	no

All languages are orthodox hardware description languages, i.e. they have special constructs to describe hardware specific properties.

UDL/I appears to be a language specifically designed for very accurate descriptions at the logic level. Due to the absence of examples and design tools, we cannot make a useful com-

parison with the other languages. The discussion below is therefore limited to ELLA, Verilog and VHDL.

Behaviour and structure is supported in all these languages. All languages have similar features for describing netlists. This is not particularly surprising, since netlists are a fairly low level feature of any design, and there are simply not very many fundamentally different ways of describing them. The use of internal signals, such as in Verilog, is a possible source of errors. Such a feature should not be present in an HDL. Behavioural constructs are also similar in all languages, with sequential and concurrent constructs supported. One of the drawbacks of a VHDL description is that parts of a design may be scattered across a file – there are separate declarations for the input/output interfaces (the 'entity' declaration) and the declarations for internal structure and behaviour ('architecture' declaration). A designer would normally keep behavioural and structural descriptions in separate files (called *configurations*) and swap between them for the purpose of simulation. Architecture declarations are not mandatory. In ELLA, behavioural descriptions are not mandatory either; it is possible in ELLA to write specifications that can be regarded as either behavioural or structural (as discussed above). Verilog structural descriptions look quite similar to those in VHDL, but they must be present within a single textual unit.

The timing model in ELLA is rather inflexible. The language deliberately does not provide pre-defined operators such as "at the positive edge of", causing very complex descriptions as the designer has to define his own functions for this purpose. Delay information can be scattered over several functions, causing more difficulty in understanding. The VHDL timing model is extremely powerful, but it is therefore hard to learn, and there are many pitfalls. Verilog's timing model is less powerful than VHDL, but compares well.

Hierarchy is supported in all languages. All languages can be simulated at different levels.

As far as syntax is concerned, it is clear that VHDL is a very rich language – it has a very large number of keywords, operators and syntax constructs. This means that the language is hard to learn, and that tool development is slow, because there are many different ways of describing the same behaviour. Verilog, by contrast, is much simpler.

The very fact that VHDL is a very rich language may make it more difficult to learn than Verilog. It also takes more time to learn to avoid the common pitfalls of VHDL. As a result, VHDL descriptions are longer and more complex than their Verilog equivalents, which is one of the reasons that Verilog has a large user base. For example, the Verilog statement that detects a rising edge of a clock signal would be declared as

```
@(posedge clk)
```

This would have to be declared in VHDL as

```
if ((clk'EVENT) and (clk = '1') and (clk'LAST_VALUE=0)) then
```

One can safely assume that the Verilog statement would allow more efficient simulation.

Currently, only VHDL has been subjected to attempts to define its semantics formally [75].

It is unknown as to whether any of the attempts were completely successful or not.

All languages use data types as a way of improving readability of design descriptions. All of the languages except Verilog allow basic types to be combined into composite types, thereby allowing the definition of complex data types. The ELLA philosophy means that no pre-defined types are available. This may lead to models that are more complex than is desirable. Verilog and VHDL have a good set of pre-defined functions, and in the case of VHDL there are large libraries of types available. Verilog's pre-defined types are at a fairly high level, allowing concise descriptions and fast simulation.

All languages allow modules to be specified, although in the case of VHDL (as discussed above) parts may be scattered across different textual descriptions. These modules can be re-used in other designs, or made available as part of a library.

The fact that both VHDL and Verilog behavioural descriptions are based upon imperative programming languages (ADA and Pascal, respectively) means that they have limited use as an input language for a formal verification tool. Formal verification tools for Verilog are currently not available. There are formal tools for VHDL [78], but these only allow a subset of VHDL to be used. ELLA appears to be more suited to formal verification, although efforts to verify the Viper chip [77] using HOL have not been entirely satisfactory; work on a restricted subset called picoELLA [76] appears to be successful.

High level synthesis tools are now available for Verilog and subsets of VHDL, for example within the CADENCE framework. None of the current approaches appears to be based on formal methods. This is not surprising, since both languages were specifically designed for modelling and simulation. Designing synthesis tools for these languages is difficult, and has taken a long time.

Recursive descriptions of structures are available in ELLA and VHDL. Recursive behaviour can be provided in VHDL, although its use is frequently discouraged by high level synthesis tools. Only ELLA regards recursion as central to the philosophy of the language.

VHDL is in widespread use, because it was adopted by the US Department of Defense, and very many people were involved in its design. Verilog is very popular in industrial and academic environments, with a user base comparable to VHDL. The ELLA user base is mainly concentrated in the UK, and its size and influence is diminishing.

It should be clear that none of the languages can claim all of the important aims and features set out at the start of this chapter, particularly with respect to the important use of functional features and associated formal verification and synthesis techniques.

We now introduce the features of the STRICT language which is intended to fill the gaps.

3. LANGUAGE DEFINITION

3.1. Introduction

This chapter introduces the STRICT language. We first summarise the requirements for the language. We then describe the features of the language resulting from these requirements:

- Design concepts – the use of blocks;
- The structure of blocks: interfaces, types, behaviour, structure and control;
- The overall design organisation.

Appendix B shows the entire STRICT grammar in the form of so called 'railroad' diagrams.

3.2. Requirements for STRICT

STRICT would have to have the desirable general features identified in the previous chapter. We briefly summarise these aims here, and also elaborate on how they could be achieved. In addition, the language should avoid what we believe are the main deficiencies of ELLA and VHDL. The main points are as follows:

- The language should be capable of describing behaviour, structure, and timing properties at different levels. It should show a clear separation between behavioural and structural descriptions, unlike ELLA, which allows them to be mixed together and interpreted according to need. We believe that behaviour and structure are two entirely different concepts, and that a designer should initially think purely in behavioural terms. Allowing mixed behavioural/structural descriptions would allow the designer to give in to the temptation to implement a structure at a very early stage of the design process. This would make it much more likely that inappropriate design choices would be made that would be very difficult to reverse later on.
- The language should allow flexible specification of timing requirements, and should be able to model both synchronous and asynchronous design features.
- In order to allow designs to be built hierarchically, STRICT should require the designer to specify his design in terms of blocks. All blocks may be made up out of sub-blocks in a recursive and hierarchical manner.

- **STRICT** should be well suited to formal verification and formal high level synthesis. The language must therefore be designed as a functional language. That is, it should express both structural and arithmetical expressions in functional form. Repetition should only be achieved through the use of recursive function calls, without the use of side-effects on global variables. In fact, **STRICT** should not allow global variables at all.
- Because of the intention to perform formal verification, it is essential to have a behavioural specification available. **STRICT** would therefore have a *mandatory specification* section, unlike ELLA and VHDL. **STRICT** would allow *optional structure* to be described, with the expectation that if the structure was absent, this could be filled in later, either by the designer or by automatic behaviour-to-structure translation (i.e. formal synthesis) tools. **STRICT** would not allow different parts of a definition to be scattered about, as in VHDL; all relevant information about a hardware module would be present within the same textual unit.
- Each level should allow simulation and verification, in conjunction with powerful tools, even if the level below was incomplete. It should also be possible to interface to other tools. For example, it should be possible for designs produced with the new language to be fabricated (it should not be just a modelling language). Although the language should be completely layout independent, it might be necessary to include the use of pragmas which would allow the designer to provide hints about certain low level design properties to the design system. Three kinds of such hints are necessary: one to specify a rudimentary form of floorplanning; one to indicate on which edge of a block a particular bus is situated; and one to collapse hierarchy caused by the use of recursion to achieve iteration.
- **STRICT** should have a clear syntax and semantics. We believe that the examples shown in the previous chapter clearly show that the syntax of both VHDL and ELLA (see again the ELLA 'Sigma' function) is too complex. **STRICT** should use familiar keywords and constructs wherever possible, in order to allow maximum readability

and understanding of descriptions in the language. So a STRICT description should not just be a set of equations, and should not use the LISP syntax.

- STRICT should support abstract data types. It should enforce a strong typing discipline (similar to the way this is done in programming languages such as PASCAL) upon the definition of busses, and require that only busses of the same type may be interconnected. This enables a certain class of errors to be detected before any layout is generated, which would speed up the design cycle. In addition, the designer should be able to specify assertions to make the design system perform extra checks on design properties which would not otherwise be performed. The language should include a basic set of operators for manipulating the standard data types of the language, including those for doing arithmetical and boolean operations.
- STRICT should allow description of generic components. This could be achieved by allowing blocks to have generic parameters. The STRICT design system should be able to expand the formal description when a particular member of the family is used. For example, it should be possible to design an n -bit register, without having to specify the actual value of ' n '; it would then be the task of the design tools to expand the definition of the register when the actual value became available.
- STRICT should encourage re-use of components, by allowing a design to be specified as a collection of self-contained modules. It should allow parts of designs to be imported from libraries, in order to prevent 're-inventing the wheel', and to be able to interface to fabrication facilities.

We now introduce the language that follows from these requirements.

3.3. Design concepts

Hierarchy is a basic part of the language. This is achieved by building a STRICT description as a hierarchy of interconnected modular blocks. The blocks are connected together via interfaces. Data abstraction is achieved by requiring each interface to be of a particular type. Interfaces are then only allowed to be connected to other interfaces of a matching type. The type specifies the representation of the information that flows through interfaces

between blocks. STRICT enforces a regime of *strong typing*. A bus transmits information of a specific type from one part of the circuit to another. A block transforms information to implement functions or to change its representation. STRICT declares the connections between the blocks but does not define the final physical layout. A particular bus may interconnect several blocks. Each block interface has an input or output attribute. An output (input) interface of a block may only be connected by a bus to an input (output) interface of another block. Each block is designed independently from any of the other blocks, and must have a behavioural section that describes functions that are to be implemented in the electrical design.

Wherever possible, language descriptions will be in a functional form, and they will use appropriate keywords to enhance readability.

3.4. The Block

3.4.1. Basics

A block defines a set of devices that transform data. Each block is identified by a block header, contains a declaration of the components within the block, and defines the design of a circuit made up of those components. Since the components are frequently themselves defined in terms of blocks, a hierarchy of components is thus created.

A block description has the following outline. As in other parts of the language, meaningful keywords are used where appropriate.

```
block <identifier> <design_parameters>
    <interfaces>
    <block_declarations>
    <block_definition>
end
```

The block identifier is used to identify a block; a particular block can be retrieved from a library of blocks so that an instance of that block may be incorporated in further designs.

3.4.2. Design parameters

One of the requirements of the STRICT language is that it should allow a block to be designed in a generic manner, for example, with respect to the number of wires in the inter-

face. This allows a whole class of devices to be designed with one description. This is achieved through the use of design parameters, which play the same role (and have a similar syntax to) formal parameters in function definitions. Design parameters play an important part in building a device as a structure amenable to formal verification, using theorem provers that are capable of performing mathematical induction.

The following are examples of block headers with design parameters:

```
block parallelfulladder (size : integer)

block traffic_light_control (light_sequence : country_code
                             number_lights : integer)
```

The parallel full adder has a design that is parameterised by the size of the operands that are to be added. The traffic light controller is parameterised by the light sequence used to signal traffic directions, and the number of lights for which signals must be produced.

3.4.3. Interface specifications

One of the STRICT requirements is the use of strong typing, popularised by programming languages such as PASCAL. The syntax for type definitions used in STRICT is clearly inspired by PASCAL, but requires optional additional elements in order to allow simulators to use these types efficiently. As explained above, types are used in the description of interfaces between components, and are also widely used in functional descriptions. An interface specification is started by the keyword **having**. For example, the following example includes declarations for interfaces to a parallel full adder:

```
block parallel_full_adder (size : integer)
    having (a,b @w,c @w : posint(size)) :
        (s @e : sum)
```

The parallel full adder block is declared with three input interfaces 'a', 'b', and 'c' of type 'posint(size)'. The output interface is identified by the identifier 's' which is of type 'sum'.

In addition to specifying the interfaces, the interface specification can optionally position the individual interfaces on particular sides of the block, using edge identifiers. This (admittedly inelegant) construct provides for another aim of the language: i.e. ensuring that

layout tools can achieve a degree of efficiency in generating silicon. The four edge identifiers, @n, @s, @e and @w are appended to those interfaces that are explicitly positioned. Any interfaces that are not explicitly positioned, are positioned automatically by the layout system.

There are occasions when a tri-state output would be used. STRICT indicates the presence of such a tri-state by simply appending an asterisk to the appropriate pin-name in the output interface.

3.4.4. Block declarations

When a design makes use of generic parameters, not all possible values may be legal. STRICT therefore includes block declarations, which describe any restrictions imposed on the generic parameters. In addition, block declarations may introduce convenient definitions and macros for subsequent use, define the types of the interfaces used in the construction of the block, and specify the functions that are implemented by the block.

An example of the header and declarations of a block describing an integrated circuit for a multiplier is as follows:

```

block multiplier(n : word_range)
    having (a,b : posint(n)) :
        (prod : posint(2*n))

inherit
    max_word_size from 'standard_library'

define
    maxmultsize = 32

    power2(n:integer):boolean ::=
        if (n < 0) then
            power2(0-n)
        else
            if ((n mod 2) == 1) then
                false
            else

```

```

        if (n == 2) then
            true
        else
            power2(n div 2)

type
    word_range ::= {is [2..max_word_size]}
    posint(n:integer) ::= { is [0..(2*n)-1]}

! Block design: A recursive multiplier for
! use in Signal Processing Applications

assert
    (2 <= n) and (n <= maxmultsize) and power2(n)

size
    10 by 10

with behaviour
    .....
end

```

The multiplier has a design parameter 'n' that specifies the size of the busses which carry the multiplicands. It has two input interfaces 'a' and 'b' and an output interface 'prod'.

The two input interfaces are of type 'posint(n)' declared as a subrange of base type integer by the type declaration section headed with the keyword **type**. The upper bound varies depending on the value of 'n'. The output interface is of type 'posint(2*n)', which is declared as another subrange of base type **integer**. The types restrict the use of the block so that it can only be connected to another block with interfaces of type 'posint(n)' or 'posint(2*n)'. The design parameter 'n' is checked for conformity when an instance of the multiplier is created for use in another design.

The 'define' statement is used to define constants and functions. Functions are examined in more detail below.

In order to allow the use of components from cell libraries, or the re-use of already defined STRICT components, the 'inherit' statement is available. In this case, it is used to retrieve the definition of the constant 'max_word_size' from a library.

The 'assert' statement restricts the design parameters of the multiplier so that 'n' is a power of 2, and in addition ensures it lies in the range 2 to 'maxmultisize', a constant defined earlier. This encourages visual checking of the design as well as providing a means for the STRICT translator to check the consistency of use of a block in a design.

The 'size' statement gives an estimation of the size of the block. This is again used as an aid to achieving efficient layout, particularly for use with floorplanning tools.

STRICT designs are commented by placing text after an exclamation mark. The comment terminates at the end of the line.

Definitions, types, size and assertions may be optionally omitted if unnecessary, but the behaviour must always be present, as explained above.

We now take a closer look at the various kinds of block declaration.

3.4.5. Inherit statement

STRICT requirements include facilities for accessing libraries, to facilitate re-use of components. The language therefore includes the 'inherit' statement, which is used to indicate the inclusion of cells, blocks, pre-defined types, functions and constants. These definitions may reside in other blocks defined within the current design file, or in completely separate design files, such as standard libraries.

An example of an 'inherit' statement is:

```
inherit  
    a, b, c from 'library'  
    x, y, z from block_identifier
```

When a 'library' or 'block_identifier' are not explicitly defined, the system defined standard design library is assumed.

3.4.6. Define statement

At this point in a block, it is possible to define new constants and functions which will be available from everywhere inside the current block. The 'define' statement can be used to define constants and function definitions.

3.4.7. Assert expression

The 'assert' statement provides a mechanism for restricting the design parameters of the block. Blocks are designed to be as general as possible. When a block is needed, a request for a particular implementation will be made. The request is made by providing actual design parameters to replace the formal design parameters in which the design is expressed. The assertion expression is evaluated. If it fails, an error exception occurs.

An example of an assertion is:

```
assert (n <= 32)
```

3.4.8. Size expression

When the design of a block is first attempted, generally only the design parameters, interfaces and behaviour are provided. When the initial STRICT notation for this block is supplied to the STRICT design system, it is possible to provide an initial estimate of the overall size of the block. This estimate can be used by some of the design tools, such as the viewer, to estimate the overall size of a component built hierarchically out of multiple sub-components. The 'size' construct is used to indicate this initial size, as follows:

```
size 10 by 8
```

The dimensions used are 'standard units'.

The final block declaration statement is the type declaration. This is now explored in the following section.

3.5. Type declaration

3.5.1. Basics

The basic types supported for design parameters are scalar or enumerated types. A scalar type can be an integer, sub-range of integers or character. Some examples:

```

type x ::= integer
      y ::= {is [1..5]}
      z ::= {is (red, yellow, green)}

```

User defined basic types are declared as part of the block declarations. They could also be inherited from a library of basic type definitions.

In addition to the provision of the basic types, 'array' and 'record' constructs are available to construct aggregate types. The notation for declaring arrays and records is very much inspired by the Pascal programming language. Some examples:

```

type singlebit ::= {is (0,1)}
      pair      ::= {s,c : singlebit}
      sum       ::= pair[n]

```

The basic type 'singlebit' is a simple enumerated type. It is then used by the type 'pair' which is a record of two variables of type 'singlebit', one called 's' and the other called 'c'. The type 'sum' is an 'n'–element array of type 'pair'. The index of the first element of an array is always number zero.

Closely connected with the type declarations are the mechanisms for implementing the types that are used as interfaces to the block. So far, the type declaration has specified what values the type will take but it has not specified how the type may be represented as a bus. This is the responsibility of the representation and conversion statements which optionally appear after the primitive type.

3.5.2. Type representations

Type representations specify the means by which the input and output interface types are to be physically supported (this is needed by most of the design tools). So far, the types will have been specified in terms of the standard types of STRICT. It is now necessary for the physical implementation of these types to be defined within the type declaration. This is done by means of two additional parts to the type declaration. The first explicitly specifies how a type will be represented in terms of an interface type called **wire**, and the second part states how the electrical patterns on wires are mapped to values associated with the type.

A wire is capable of carrying one bit of information. To carry more complicated information, arrays of wires will be required, and they can be thought of as typed busses.

Consider the following example, in which the type 'posint' and 'colours' are both used in the block interface.

```
type
  posint(n:integer) ::=
    { is [0..(2**n) - 1]
      represented by posints: wire[n]
      with mapping
        Sigma(1,n,posints)
    }
  colour ::=
    { is (red, green, amber)
      represented by colours: wire[2]
      with mapping
        if (colours[0] == low) then
          if (colours[1] == low) then
            green
          else
            red
        else
          amber
    }
```

The two types have been defined to take a particular range of values; 'posint' is a sub-range of integers up to some maximum based on 'n' which is assumed to be non-zero, and 'colours' the three colours 'red', 'green', and 'amber'. The representation has defined a physical definition in terms of wires. That is, 'posint' will be represented by n-bit wires and 'colours' by 2-bit wires.

The values of the type 'posint' are obtained from the wire specification (i.e. 'posints') by calling the function 'Sigma' (see the next section) with arguments '1', 'n', and 'posints'.

The type 'colour' however just uses an expression to determine how the values 'red', 'green' and 'amber' are represented in two wires.

3.5.3. Representational functions

The 'posint' example of the previous section shows that there will frequently be types where the mapping from the physical representation to the range of values of that type is relatively complicated. It may well depend upon functions (or operations) that are not immediately available in STRICT. Therefore, the facility is provided for a designer to define his own functions to map the physical representation onto the range of values.

General purpose functions can be defined at the head of blocks and implementations, and also as part of behavioural specifications.

Representation functions are defined in a similar manner to other functions except that they appear after the type declarations they refer to.

For example, the functions 'Sigma' and 'Decode' can be defined as follows:

where

```
Sigma(lower, upper: integer,
      w: wire[*]) : integer ::=
    if (upper == lower) then
      Decode(w[lower-1])
    else
      Decode(w[lower-1]) +
        2 * Sigma(lower+1, upper, w)

Decode(w: wire) : integer ::=
    if (w == low) then
      0
    else
      1
```

This concludes our definition of types and associated functions. We now turn our attention to the behavioural declaration section in a block.

3.6. Behavioural specifications

3.6.1. Basics

The aims of the STRICT language require that behavioural specifications are compulsory. Ensuring that every block has a behaviour means that we can achieve another aim of the STRICT language: that a design can always be checked, no matter how incomplete it is. A simulator or formal verification tool can check the behaviour of a given block, with the implementation of that block and the behaviour of all the sub-blocks used in that definition.

The design of this part of STRICT has probably been the most difficult, as designers may be very reluctant to specify beforehand, in a formal manner, the behaviour of the block that they are about to design. Once the block had been designed, they are usually prepared to provide a model of their design for simulation, but quite often by that time, the design is already different from what they had set out to design initially.

A designer using STRICT however will be required to provide a behavioural specification, before he attempts to implement the block, thereby enabling a check to be made on the design to ensure it still matches the specification provided at the beginning of the design process.

3.6.2. Example

We again illustrate typical behaviour with an example. Previously, a block declaration for a multiplier was provided. The (missing) behavioural specification could be as follows:

```
with behaviour
  whenever
    change(a) or change(b) :
      within (10 * sigbits(max(a,b)))
        set prod = a * b;
  where
    sigbits(x : integer) : integer ::=
      if (x <= 1) then
        1
      else
```

```

1 + sigbits(x div 2)
max(x,y:integer) : integer ::=
    if (x > y) then
        x
    else
        y

```

The specification of the block, denoted by the keyword **behaviour**, describes the intended function of the multiplier. The pre-condition for the multiplication is that either input identifiers 'a' or 'b' must change before the appropriate action is performed.

Given the pre-condition, the specification provides the post-condition for the output interface 'prod' after the multiplication has occurred. The post-condition includes a temporal expression denoted by the keyword **within**. The temporal expression specifies a maximum possible delay from the start of the multiplication of the two multiplicands to the output of the result in 'standard time units'. The result of the multiplication is specified with the 'set' clause, using the integer multiply operation.

Apart from the features shown in the example, the language also allows for the declaration of state variables, initial values of state variables, and invariant expressions on state variables.

3.6.3. States

Most circuits require the concept of *state*. That is, they use a knowledge of what has happened in the past, to calculate their next set of outputs.

To capture this concept in a behavioural description involves either the use of state variables, to record the past values of interest, or the ability to inspect past histories of an input or output, and make use of the values found there.

It was decided to use the state variable approach in STRICT, because it was easier to implement. Examples of state declarations are as follows:

```

state idle  ::= boolean
owner      ::= (p1,p2)
mem        ::= char[n]

```

In this example, three state variables are identified. The first, 'idle', is of type **boolean**. The second, 'owner' is an enumerated type. The third, 'mem' is an 'n' element array of type 'char'. It is assumed that both 'n' and 'char' will have been defined previously.

3.6.4. Initial statements

Once a state variable has been declared, it is often useful to specify what the initial value of that variable must be. The 'initial' statement specifies what particular values state variables must start with. Any non-structured type may be given an initial value and an array of non-structured types may also, in a single statement, be assigned a starting value. Any record based types must have their fields assigned separately.

The following are typical of the use of the 'initial' statement. The example uses the state variables declared above.

```
initially idle = true  
          mem = 0
```

The boolean identifier 'idle' is given the initial value of **true**. Each element of the array 'mem' is given the value of 0 in the second assignment. The only way to initialise individual elements of an array to different values, is to initialise each element individually.

3.6.5. Invariants

Within the behavioural specification it may be possible to identify particular states that the block must never reach. These states may specify particular input or output values or combinations of values. **STRICT** allows for these states to be identified and therefore checked.

An invariant can be provided within the behavioural specification which must remain true while the behavioural specification is being exercised. If the invariant is found to be false, an error exception occurs.

For example, when designing a traffic light controller, it would be crucial not to allow both sets of lights to be 'green' at the same time. This could be specified with the following invariant.

```
invariant not ((road1 == green) and (road2 == green))
```

'road1' and 'road2' are the output interfaces to the traffic light controller and specify what colour the attached light should show. This invariant therefore specifies that at all times, at least one of the two roads should not be showing a 'green' light.

3.6.6. Simulator Timing model

The basic principles of timing description are presented by Aylor [73]. A series of statements should be presented. The linear ordering of these statements defines the normal sequence of processing. Statements should be combined with boolean conditions or guards to control the execution of each statement. All statements with conditions executing to **true** should be executed. Conditions should be allowed to have states. States should be evaluated as part of the conditions, to let designers create any desired sequence of operations.

STRICT conforms to these requirements, but additionally allows the specification of exception conditions. The formal simulation model is defined as follows [71].

Behaviours associated with blocks are executed in response to the arrival of signals from the outside world. A behaviour is defined as a set of at least one causal event 'E' and an associated assignment set 'A':

$$B = \{(E,A)\}+$$

Each assignment set 'A' consists of a number of elements: a selection condition 'C' (which determines the conditions under which the assignment can occur), an assignment function set 'S' (which must contain at least one element), and a temporal constraint 'T':

$$A = \{C,\{S\}+,T\}+$$

The temporal constraint defines the ambiguity delay after which the assignment can occur, and an associated guard 'G':

$$T = (w, G), \quad 1 \leq w$$

The simulator will of course have to decide the precise moment at which the assignment will have to be scheduled within the specified interval. The current version of the simulator performs the assignment two thirds of the way through the interval.

'G' is defined as

$$G = (f, U)$$

where 'f' is a guard period during which the assigned values must not change, and 'U' is a causal event which can pre-empt the completion of the block. 'G' may be empty.

The STRICT description will specify all of these elements for each block in the hierarchy. Let us now consider the elements of the timing model in more detail.

The assignment set (i.e. the behaviour) is specified by a sequence of actions. The execution of an action is guarded by a cause from the event set 'E' which must evaluate to **true** before the action is executed. In addition, there may be some additional conditions (from the set 'C') that are associated with the cause (typically, state variables), and they too must evaluate to **true** before the action is executed. All the causes within the specified behaviour are evaluated concurrently with no implicit priority being assumed. Once a particular cause and associated conditions, if any, has evaluated to **true**, then the specified action is taken.

A temporal constraint, i.e. an ambiguity delay, specifies the time after which a particular effect will have been achieved. This effect can be held for a certain duration, or until some other exception occurs which then interrupts the effect, leaving the block in its new state.

3.6.7. Selection and Causal Guards

The causal guard is responsible for identifying when an event occurs, so that the appropriate action may be performed. Events are in general the changing of values on input interfaces.

For example, an input interface called 'clk' may be used as a causal guard by requesting that the cause becomes true when the 'clk' line rises. STRICT has a standard type called 'clock' which can be parameterised to specify how long the line will be high, and how long the line will be low. Two standard values, called **high** and **low**, are available which would correspond to the particular technology's way of representing a bit. There is also a standard function called 'change' which takes any input interface as an argument, and returns an event as soon as the argument changes its value. An event is very similar to a boolean, in that it can take the values **true** and **false**, but in addition a time is also associated with the truth value. This time indicates the exact moment that the event occurred. A suitable clause to detect the rise of a clock would be

```
change(clk) and (clk == high)
```

It is possible in a causal guard to form the conjunction and disjunction with any other boolean expression. Therefore in the above example, the conjunction is formed with the expression '(clk == high)' to check if the clock has just risen.

Along with the causal guard, it is also possible to associate selection guards. These are used to make additional checks on the state of the block before a particular action is allowed to occur. They are represented by boolean expressions. For example, the following is a selection guard.

```
(road1 == green) and (~car)
```

This condition guard compares the value of 'road1' with the constant 'green' and then forms the conjunction with the negation of the boolean identifier 'car'.

3.6.8. Ambiguity Time Delays

Before the action requested by the causal and selection guards is actually performed, it is possible to specify a time delay after which the results of that action will be guaranteed to be stable. If another block should try to make use of these results before the ambiguity delay has passed some form of race condition will be indicated to the design system.

The ambiguity delay is specified using the within expression. For example,

```
within (17)
within (10 * sigbits(max(a,b)))
```

The first example specifies a fixed delay of 17 'time units'. The second specifies a variable time delay which is based on the value of a couple of user defined functions. i.e. 'sigbits' and 'max'.

3.6.9. Effects

The effect of the action is performed by the 'set' statement. This specifies which identifiers are to be assigned new values. For example,

```
set
    prod = a * b
    idle = true
```

In the example above the identifier 'prod' is given the value of the identifier 'a' multiplied by 'b'. Then 'idle' is given the value **true**.

3.6.10. Duration time delay

Once the ambiguity delay for an effect has elapsed, that value is usable by any other part of the behavioural description. It is sometimes required, however, that an effect should last for a minimum period of time after the end of the ambiguity delay. This situation is provided for in the provision of a duration delay associated with each ambiguity delay, called 'G' above – an indication that the previous effect must last for a certain minimum period of time. The 'for' clause specifies the appropriate duration time delay. For example, it may be used as follows:

```
for 10
for 25 * cycle(clk)
```

In the first example, a duration delay of 10 'standard time units' is specified. The second example, assuming that the identifier 'clk' has been defined to be a clock input interface, specifies that the duration delay is twenty five times the period of the clock identifier 'clk'. It is also possible for users to define their own functions.

3.6.11. Exception conditions

We now describe the features of the set called 'U' above.

Once a duration delay has been provided, if it is executed, then it guarantees that the effect it follows will last for the specified duration without any interruption. It is sometimes necessary, to provide the facility for interrupting a duration by the arrival of a particular event. This is called an exception condition and the 'unless' clause is provided to provide this facility. The following example demonstrates how it can be used.

```
set
    road1 = red
    road2 = green
for 10 * cycle(tick)
unless rise(clk) and (~car);
```


This examples specifies an effect of setting two identifiers 'road1, and 'road2' to 'red' and 'green' respectively for a certain duration. This duration can however be interrupted before it is finished if the exception condition becomes true. The exception condition becomes true if the input interface 'clk' rises and the boolean identifier 'car' has the value **false**.

Occasionally, it is required to provide an infinite duration delay which can be only interrupted by certain exception conditions. Instead of specifying an infinite duration explicitly, the 'until' clause is used. For example, consider the example above where the two identifiers 'road1' and 'road2' are being given the values 'red' and 'green' respectively. Instead of these values only being present for a specified duration, it might be necessary to assign them forever, but with the same exception conditions being present. This may be done as follows:

```
set
    road1 = red
    road2 = green
until rise(clk) and (~car);
```

Once an exception condition is true, the action that was being performed is terminated, and the causal and condition guards are once again checked for a match. The block stays in the new state specified by the interrupted action.

3.6.12. Behavioural functions

In the examples given so far, a number of standard operations have been provided for use within a STRICT design. In accordance with the STRICT functional philosophy, the language uses functional forms as its main specification mechanism. The language allows the designer to define his own functions at the end of the behavioural specification, in particular those functions that have already been used in the rest of the specification, plus any additional ones as required.

For example, the user defined functions called 'rise' and 'max' have been used but not defined. Therefore they must be provided at the end of the behavioural specification, preceded by the keyword **where**:

where

```
rise(x:clock):boolean ::=
    change(x) and (x == high)

max(x,y: integer): integer ::=
    if (x>y) then
        x
    else
        y
```

'rise' returns the value **true** whenever there has been a change on its parameter, and the current value is high. 'max' returns the maximum of its two parameters in the usual manner.

Both functions specify their argument types and the single result type of the function. There then follows the expression which calculates the value of the function using the supplied arguments. Any type defined by the user or provided for in the implementation of STRICT can be returned as the result of a function.

This completes the description of the behavioural specification of a STRICT block. We now turn our attention to structural specifications.

3.7. Structural specification

One of the requirements of STRICT was that it should be capable of describing structure. As explained before, structural description is optional.

3.7.1. Basics

The structural specification refers to the way in which a block is implemented in terms of other blocks, and how these blocks connect with each other. It is also able to suggest, via hints within the definition, how the constituent blocks should be arranged.

The block definition is again made up of a number of smaller sections. Each section is responsible for part of the definition, and is preceded by an appropriate keyword.

3.7.2. Use of recursion

One of the requirements of STRICT is to use recursion as the only means of achieving iteration. This recursion is most noticeable in the implementation section. It is controlled by

the generic design parameters of the block. When an implementation requires that a recursive call on the block being defined is necessary, it will modify the design parameters it was initially given, usually making them simpler, before requesting an instance of that block. For example, suppose the implementation of a particular block depended upon the generic design parameter 'n'. If 'n' was less than or equal to 1 then one particular implementation was required, otherwise another was required. It would be presented as follows:

```
use structure
  (n <= 1) :
    { ... implementation version 1 ... }
  (n > 1) :
    { ... implementation version 2 ... }
```

When the design system is asked for a particular implementation by the binding of the design parameters to actual values, it evaluates the guards in turn. The first guard that is found to be true then specifies the particular implementation to be used. If none of the guards is true, an error exception occurs.

3.7.3. Inherit statement

The 'inherit' statement, first introduced in the block declarations section, when used in this particular context, allows for the inclusion of other blocks, thereby encouraging re-use of existing components. All blocks used within a particular implementation, must either exist within the current design file or be explicitly inherited from the design file in which they were defined. If an 'inherit' statement is used, but no explicit design file or library is requested, then the standard design library is assumed.

3.7.4. Instancing components

Before a block is used within an implementation, it requires a local name to differentiate it from other blocks. For example, the implementation of a recursive multiplier requires the following blocks.

```
instance
  lowmult,midmult1,midmult2,highmult: multiplier(multsize)
```

```
cp : carrypropagateadder(n)
pa : paralleelfulladder(n)
```

Six blocks are declared, four of which have the same block identifier. The identifiers 'n' and 'multsize' are the actual design parameters.

3.7.5. Hierarchy flattening

In order to increase efficiency in the case where a component is defined recursively with many levels of recursion, it is possible to add the optional keyword **collapse** as a last parameter in the parameter list of a component. For example, the statement

```
instance
    reg: register(16, collapse)
```

would indicate to the layout software that the register should be flattened. It is again hoped that such a construct would be unnecessary in the future.

3.7.6. Placing the components

STRICT allows the designer to specify some form of placement strategy. It provides the 'place' statement which may be used by the layout system to place blocks. If a placement is not suggested, then the layout system will attempt to provide an appropriate placement strategy of its own.

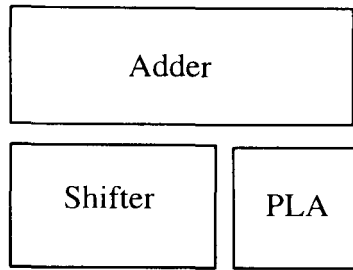
STRICT makes the assumption that all layout blocks are rectangular in shape. This means that only two placement operators are provided: ';' which corresponds to 'right-of/left-of' and '/' which corresponds to 'above/below'.

An example of a placement statement is

```
place
    Adder/Shifter;PLA
```

That is, the block identified as 'Adder' is to be placed above the remaining blocks. Of the remaining blocks, 'Shifter' is to be placed to the left of 'PLA'. Brackets can be used to group together blocks that are to be treated as one.

This would arrange the three components as follows:



Any placement that can be described by a tree can therefore be described in STRICT. The 'place' statement is optional, in keeping with the requirement that tool specific keywords must be optional. If it is omitted, the floorplan is determined automatically.

3.7.7. Module generator interface

STRICT is required to interface to module generators, again in order to allow efficiency. As a result, the language allows a second form of the 'use structure' statement, which has a string between the two keywords. For example, the statement

USE 'PLA' STRUCTURE

would indicate that a call to a PLA generator was required at this stage of the layout process.

3.7.8. Interconnecting the components

The final two sections of a block definition are concerned with connecting the instanced sub-blocks together and then connecting the block interface to the appropriate sub-components.

Interconnection of the sub-blocks requires the 'using' statement. For each block, it is necessary to identify the actual input interface to be used. This may simply be the input interface of the block being defined, or the output interfaces of one or more of the sub-blocks being interconnected.

STRICT, being strongly typed, requires that the actual interface parameters agree 'by name'. There may be occasions when two or more interfaces need to be combined together, or one interface needs to be split into several, before connection to a sub-block is possible. A standard set of functions is available to provide the basic facilities. More complicated

methods may be designed by defining suitable functions which operate on the appropriate interfaces.

The following examples illustrate some of the above points.

using

```
cp(join(pabus(n) | pa.s[1].carry, tail(pa.s)),
    mostsighalf(highmult.prod))
pa(mostsighalf(lowmult.prod), midmult1.prod,
    midmult2.prod, leastsighalf(highmult.prod))
lowmult(leastsighalf(a), leastsighalf(b))
highmult(mostsighalf(a), mostsighalf(b))
midmult1(leastsighalf(a), mostsighalf(b))
midmult2(mostsighalf(a), leastsighalf(b))
```

There are six sub-blocks in this example which are being interconnected. The functions 'mostsighalf' and 'leastsighalf' are responsible for breaking the type representation of their arguments into two equal halves. (If the representation cannot be broken into two identical halves, an error exception occurs). The standard function 'join' indicates that the type representations of the arguments after the bar are being concatenated, and then coerced into the appropriate type representation indicated before the bar. Output interfaces of sub-blocks are indicated by the use of the block identifier followed by the output interface required.

The syntax for the 'using' statement has deliberately been chosen to look like a function call.

3.7.9. Connecting the interface

The final section of a block definition involves specifying how the output interfaces are connected to the constituent blocks. Again it may be necessary to coerce the type representations into the appropriate form.

The make statement is used as follows.

```
make prod ::= join(posint(2*n) |
                    leastsighalf(lowmult.prod),
```

```
pa.s[1].sum,  
cp.s)
```

There will be one statement for each output interface in the block declaration.

3.7.10. Control modelling

One aim of the STRICT language not discussed so far is the modelling of asynchronous hardware. Asynchronous hardware uses the data flow paradigm. There is no global clock. Data is passed from one subsystem to the next, whenever computation is complete. The issue here is how the data transfer is controlled, that is, how the different subsystems within a block interact.

In some design styles, particularly those aimed at the implementation of digital signal processing algorithms, data passed from one component to the next uses point-to-point connections between the two. In a network formed by components interconnected by connections of this type, the structure of the network implies the data flow, and hence separate control hardware is not necessary.

Whilst it is possible to produce a wide range of systems using data flow methods, many architectures cannot be efficiently implemented without a separate controller, since they use a limited set of busses and general purpose arithmetic and other functional blocks each of which may be used for many different data transfers. Typically the CPU of a microprocessor is designed in this way, with a datapath being used for different data and with different operations during the course of a single instruction.

In this case, the structure of the data path is insufficient to describe the behaviour of the system without an explicit view of the function of the control. In STRICT, a mechanism has been provided to capture the control flow. This allows the synthesis of a range of controller architectures such as PLA's, microprogrammed memory, or self-timed architectures, and it also allows display of the control flow in a graphical form.

A block with a control section is indicated by the keyword **asynch** which prefixes the **block** keyword, and a special section within the block started by the keyword **control**. Such a block is assumed to have in addition to its explicit inputs and outputs, two control lines (cur-

rently known as 'req' and 'ack'), which may only be manipulated by the behavioural and 'control' section of the description. Two other keywords, **wait** and **signal**, manipulate control signals which are assumed to be single bits, capable of representing 1 or 0. **Wait** is assumed to detect a rising edge on the control signal specified.

We now show an example: an asynchronous divider. The divider is a block which performs division by repeated subtraction. It requires a number of registers, a subtraction unit, and a test to see if the end of the computation has been reached. During the computation, handshake signals cause the proper sequencing of signals.

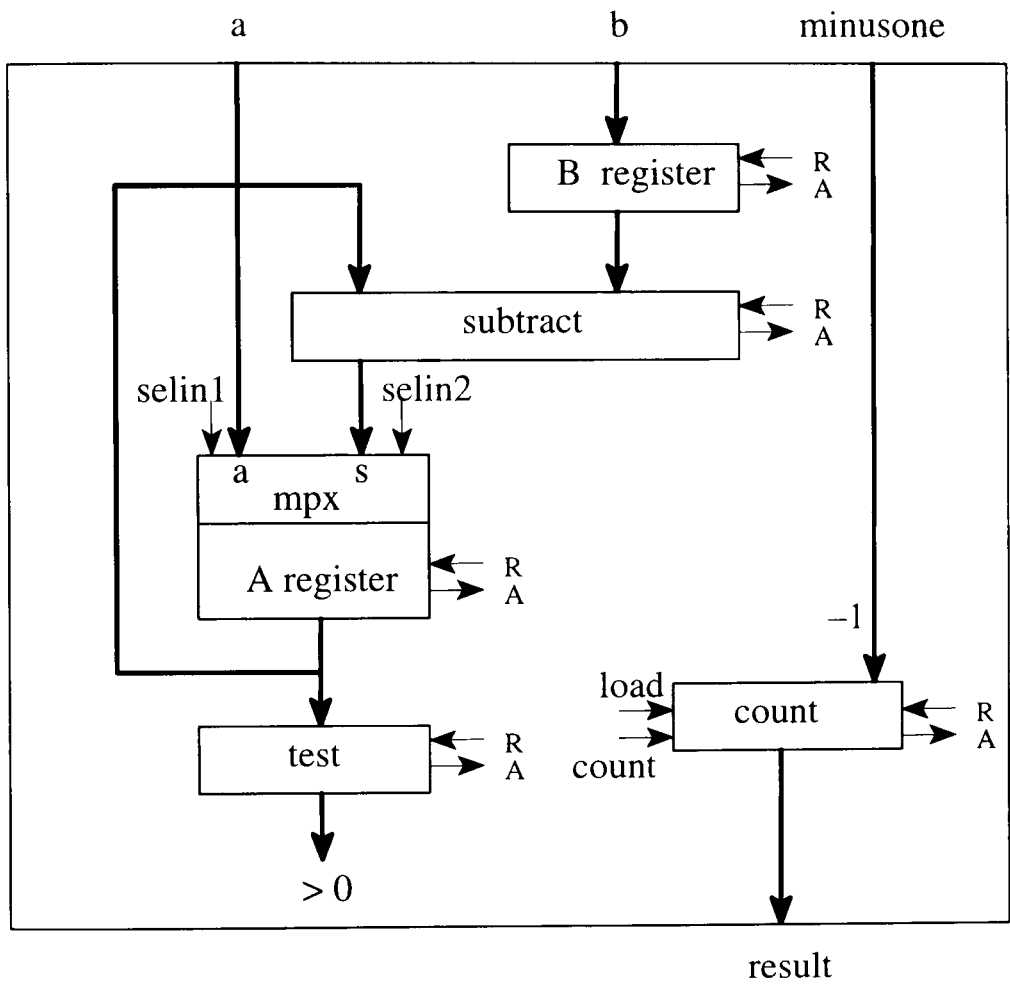


Fig. 3. Divider module

An overview of the divider is shown in Fig. 3.

The number to be divided is initially loaded into register 'A', which requires setting the control signal 'selin1'. The divisor is loaded into register 'B'. This happens only once, and register B is not changed during the computation. Register 'A' is used to hold the intermediate results of the repeated subtraction. During the computation, the control signal 'selin2' is set to ensure that 'A' is updated from the subtraction module, not from the input port (this is the reason why a multiplexer is present). The 'test' module will indicate when 'A' contains a negative value, which means that the computation is finished.

The circuit also contains a counter, which holds the number of iterations performed by the circuit (i.e. the quotient). It is initially loaded with the value -1 , because once the value in 'A' has become negative, the count process has gone one step too far. The counter has two control signals, one for loading -1 into it, and one for counting.

All sub-modules have a request/acknowledge pair.

The STRICT description is as follows. Line numbers have again been attached in order to make explanation of the description easier.

```

1/ ASYNCH BLOCK divider
2/   HAVING (a,b,minusone : number):
3/           (result : number)
4/ INTENDED BEHAVIOUR
5/           WHENEVER
6/           WAIT divider:
7/           WITHIN (10)
8/           SET result = a DIV b
9/           SIGNAL divider;
10/ USE STRUCTURE
11/   {
12/       INSTANCE breg: reg
13/           areg: muxreg
14/           sub: subtract
15/           test      : greaterthanzero
16/           count: counter

```

```

17/          CONTROL
18/          WAIT divider:
19/          SIGNAL breg, areg, count, areg.selin1
20/          SET count.load = 1
21/          count.incr = 0
22/          WAIT breg, areg, count, areg.selin1:
23/          SIGNAL test
24/          SET areg.selin1 = 0
25/          WAIT areg, count, areg.selin2:
26/          SIGNAL test
27/          SET areg.selin2 = 0
28/          WAIT test:
29/          SET ack = ~test.output,
30/          sub.req = test.output
31/          WAIT sub:
32/          SIGNAL areg, count, areg.selin2
33/          SET count.load = 0
34/          count.incr = 1
35/          USING areg(a,sub.s)
36/          test(areg)
37/          sub(areg,breg)
38/          breg(b)
39/          count(minusone)
40/          MAKE result ::= count
41/          }
42/  END

```

The explanation of the description is as follows.

The behaviour (lines 5–9) specifies that after the block is activated (line 6), the output called 'result' will contain the quotient of inputs 'a' and 'b' after 10 time units. The next block in the chain (not shown in the picture, but connected to port 'result'), is then activated (line 9).

Of the 'use structure' section, only lines 17–34 are of interest here (the other lines just declare the components and connect them).

Lines 18–21 specify what happens when the divider is first activated, as indicated by line 18. Registers 'A' and 'B', the counter, and the multiplexer are all involved, so their control signals are set (line 19). Since the multiplexer has explicit control signals, the name of the signal must be provided. The other blocks just use the standard req/ack pair. In lines 20 and 21, the 'load' and 'count' inputs to the counter are set to appropriate values.

Once this been completed (line 22) it is checked whether 'A' is negative already (i.e. we should stop straight away) in line 23, and the 'selin1' line is reset in line 24.

The results of the 'test' module (line 28) causes the subtraction module to be activated if the contents of 'A' were still positive (line 30), and completion to be signalled to the outside world if the contents of 'A' were negative (line 29).

The remaining two sections (lines 25–27 and lines 32–34) control the operation of the repeated subtraction loop.

The whole control scheme can conveniently be drawn in the form of a Petri Net. The viewer subsystem can do this automatically. The result is shown in the chapter on the viewer.

3.8. Design organisation

A complete STRICT design is in general organised as a sequence of complete block declarations, preceded by a request for the top level block, i.e. the entire chip design. This is the root of the design hierarchy tree.

An example format of a complete design is as follows:

```
build
{ instance m:multiplier(16)
  using m(ain,bin)
  make cout ::= m.prod
}

given
  block multiplier..... end
```

```

block carrypropagateadder..... end
block parallelfulladder..... end
.....

```

The top level block is responsible for specifying what device is actually being requested. Its form is exactly the same as before, except that the input and output interfaces will not exist explicitly and that all design parameters must be constants or constant expressions. All input and output interfaces will correspond with input and output pads of the appropriate type and number. That is, if an input (output) interface is requested, an input (output) pad is used.

In addition to the pads explicitly requested by the design, extra pads will also be placed by the layout system, to handle power and ground connections, and any other outside connections that are required.

This completes our description of the language. We now illustrate the language with a number of examples.

3.9. Examples

This section shows some typical examples of hardware modules, coded in STRICT. They have been chosen with a number of criteria in mind:

- To demonstrate the use of the more important features of the language;
- To demonstrate (later on) the use of some of the tools;
- To describe typical design examples widely used in the literature.

In order to facilitate explanation, line numbers have again been inserted at the start of lines.

3.9.1. Library cell

This example shows how to include a basic cell from the design library, using the 'inherit' statement. The block, called 'inverter', has a single bit input called 'a' and a single bit output called 'out' (lines 2–3). The block inherits 'ntg' from the system library (line 4), declares its behaviour (lines 5–10), instances a single version (line 13), connects the single input port of 'x' to port 'a' (line 14), and connects the output port of 'x' to port 'out' (line

15). The behaviour essentially specifies that the output is set to the inverse of the input after 4 time units (lines 8–10). The description relies on knowledge that the library cell has an output called 'qb' (line 15).

```
1/ BLOCK inverter
2/   HAVING (a: WIRE) :
3/           (out: WIRE)
4/   INHERIT ntg from '$cells$libcmos'
5/   INTENDED BEHAVIOUR
6/     WHENEVER
7/       change(a):
8/         WITHIN (4)
9/           SET
10/            out = ~ a;
11/   USE STRUCTURE
12/   {
13/     INSTANCE x:ntg
14/     USING x(a)
15/     MAKE out ::= x.qb
16/   }
17/ END
```

The structural picture of the resulting block is shown in Fig. 4. Note that it is not necessary to know what the input port of 'x' is called.

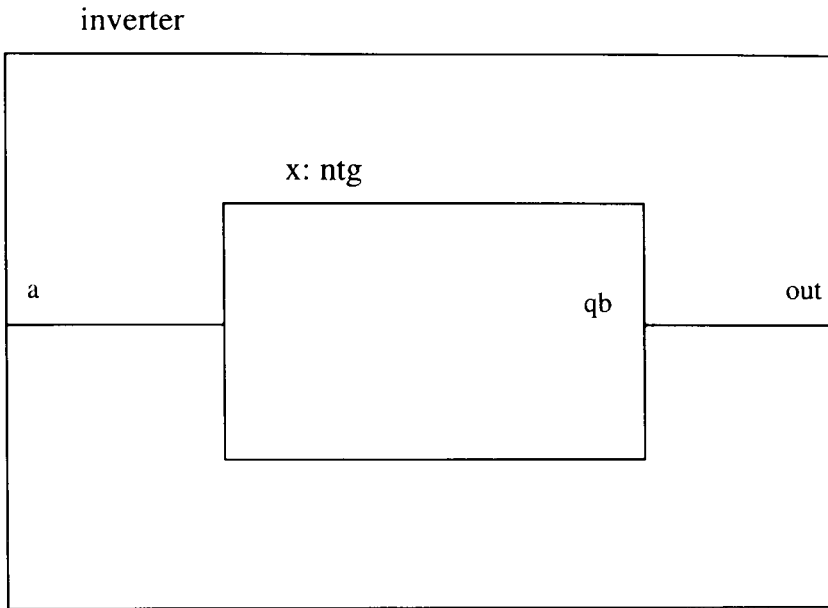


Fig. 4. Structure of inverter

3.9.2. Half adder

The half adder is a frequently used example in the literature. It takes two single bit inputs, (called 'x' and 'y', line 2) and produces a 'sum' and 'carry' bit as output (line 3). It is implemented using three AND gates, one OR gate and two inverters (declared at lines 26–28). The example shows a straightforward composition (lines 30–38) and specification (lines 17–22), and uses a type called 'bit' (lines 5–9), which uses a local function 'decode' (lines 11–15) to map voltage patterns into bits.

```

1/      BLOCK hal
2/          HAVING (x, y: bit):
3/              (sum, carry: bit)
4/      TYPE
5/          bit ::=
6/              { IS [0 .. 1]
7/                  REPRESENT BY b: WIRE
8/                  WITH MAPPING decode(b)
9/              }
```

```

10/          WHERE

11/          decode(w: WIRE): integer ::=
12/              IF (w == low) THEN
13/                  0
14/              ELSE
15/                  1

16/          INTENDED BEHAVIOUR
17/          WHENEVER
18/              CHANGE(x) OR CHANGE(y):
19/              WITHIN (15)
20/              SET
21/                  sum = (x + y) MOD 2
22/                  carry = (x + y) DIV 2;
23/          USE STRUCTURE
24/          {
25/              INSTANCE
26/                  n1, n2: inverter
27/                  a1, a2, a3: andgate2
28/                  o1: orgate2
29/              USING
30/                  a1(x,y)
31/                  a2( x,n1.out)
32/                  a3(n2.out,y)
33/                  n1(y)
34/                  n2(x)
35/                  o1( a2.out, a3.out)
36/              MAKE
37/                  carry ::= a1.out
38/                  sum   ::= o1.out
39/          }
40/          END

```

The resulting structural view is as shown in Fig. 5.

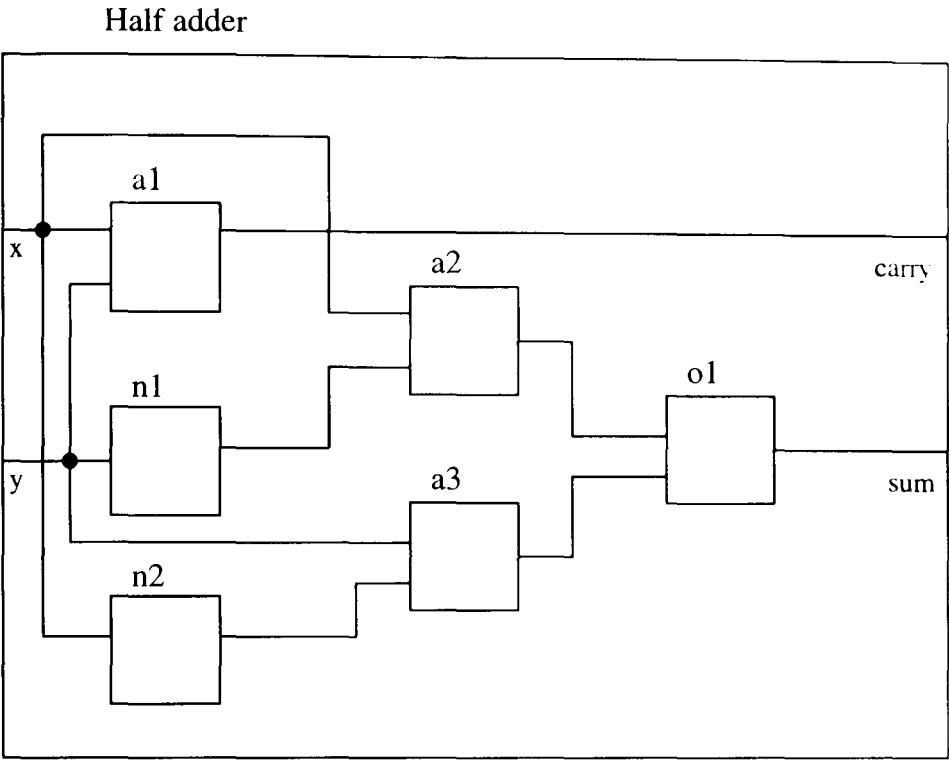


Fig. 5. Structure of half adder

3.9.3. Full adder

As an example of introduction of a complex hierarchy, the full adder uses two half adders (declared in line 15, and defined above) plus an OR gate. It takes three inputs (two data bits and a carry bit, defined in line 2), and produces a sum and carry bit as output (line 3). The actual composition is similar to the half adder example. For example, 'hb' (line 15) takes as its inputs full adder ports 'x' and 'y' (line 18). The behaviour is defined in lines 7–11. This is a typical behaviour of a piece of combinatorial logic: whenever any of the inputs change (line 7), some time later (line 8) the outputs change with values dependent upon the inputs (lines 10 and 11). The example also shows how types can be inherited from other blocks (line 4).

```
1/      BLOCK full
2/      HAVING (x, y, cin: bit):
```



```

3/                (cout, sum: bit)
4/    INHERIT bit FROM hal
5/    INTENDED BEHAVIOUR
6/        WHENEVER
7/            CHANGE(x) OR CHANGE(y) OR CHANGE(cin):
8/                WITHIN (40)
9/                    SET
10/                        cout = (x + y + cin) DIV 2
11/                        sum = (x + y + cin) MOD 2;
12/    USE STRUCTURE
13/        {
14/            INSTANCE
15/                ht, hb: hal
16/                o: orgate2
17/            USING
18/                hb(x, y)
19/                ht(hb.sum, cin)
20/                o( ht.carry, hb.carry)
21/            MAKE
22/                cout ::= o.out
23/                sum ::= ht.sum
24/        }
25/    END

```

A diagram of the structure is shown in Fig. 6.

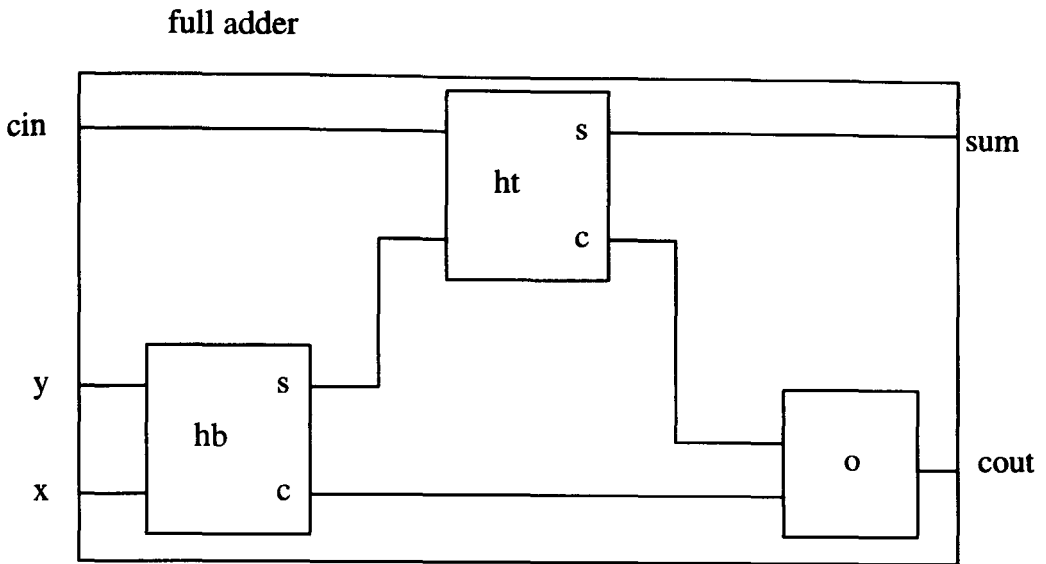


Fig. 6. Structure of full adder

3.9.4. n-input OR gate

As an example of parameterisation, we show a design for a general n-input OR gate. This is indicated by the presence of the formal parameter 'n' on line 1. The input bus (called 'in' on line 2) is therefore an n-bit one, the value of 'n' being determined by outside factors.

The behaviour shows the use (at line 10) of a locally defined function called 'orarray' (defined at lines 12–16), which recursively goes through the individual wires of the input bus whilst OR-ing the bit values held on them together. This function exhibits typical recursive behaviour: if the bus which is the actual parameter only has one wire, it returns the bit value and stops the recursion (line 13), otherwise it returns the bit value of the lowest order bit and recursively ORs it with the rest of the bus (line 14), using the pre-defined 'tail' function.

The structure shows the use of the 'case' statement in recursive implementations. When the value of 'n' is smaller than 4, the implementation is simple. For example, in the case where 'n' equals 2 (lines 21–24), 'n' equals 3 (lines 26–29), or 'n' equals 4 (lines 31–34) just an OR gate with the appropriate number of inputs is used. When 'n' is greater than 4, the gate is recursively implemented as a tree of or gates. In this case (lines 37–39) three

blocks are used: a 4-input OR gate, an $(n-4)$ -input OR gate (which is the recursive part), and a 2-input OR gate to deal with the results of the previous two structures. A diagram of the structure when 'n' is greater than 4 is shown in Fig. 7, which shows the structure plus the number of wires in the busses. The 'using' section (lines 40–44) uses the 'head' and 'tail' functions which are usually required in recursive implementations, because the actual value of 'n' is unknown. For example, the call 'tail(tail(tail(tail(in))))' produces a bus which contains the least significant $(n-4)$ wires of 'in'.

```

1/      BLOCK orgt(n: integer)
2/          HAVING (in: posint(n)):
3/              (out: bit)

4/          INHERIT bit FROM hal
5/              posint FROM Register

6/          INTENDED BEHAVIOUR
7/              WHENEVER
8/                  change(in):
9/                      WITHIN(10)
10/                         SET out = orarray(in);

11/         WHERE
12/             orarray(w:WIRE[*]):boolean ::=
13/                 IF (bussize(w) == 1) THEN decode(w[0])
14/                 ELSE decode(w[0]) OR orarray(tail(w))
15/             decode(w: WIRE): integer ::=
16/                 IF (w == low) THEN 0 ELSE 1

17/         USE STRUCTURE
18/             (n == 1):
19/                 { MAKE out ::= in
20/                     }
21/             (n == 2):
22/                 { INSTANCE o: orgate2
23/                     USING o(head(in),tail(in))

```

```

24/             MAKE out ::= o.out
25/         }
26/     (n == 3):
27/         { INSTANCE o: orgate3
28/             USING o(in[0], in[1], in[2])
29/             MAKE out ::= o.out
30/         }
31/     (n == 4):
32/         { INSTANCE o: orgate4
33/             USING o(in[0], in[1], in[2], in[3])
34/             MAKE out ::= o.out
35/         }
36/     (n > 4):
37/         { INSTANCE o2: orgate2
38/             o4: orgate4
39/             og: orgt(n-4)
40/             USING o2(o4.out, og.out)
41/             o4(head(in), head(tail(in)),
42/                 head(tail(tail(in))),
43/                 head(tail(tail(tail(in)))))
44/             og(tail(tail(tail(tail(in)))))
45/             MAKE out ::= o2.out
46/         }
47/     END

```

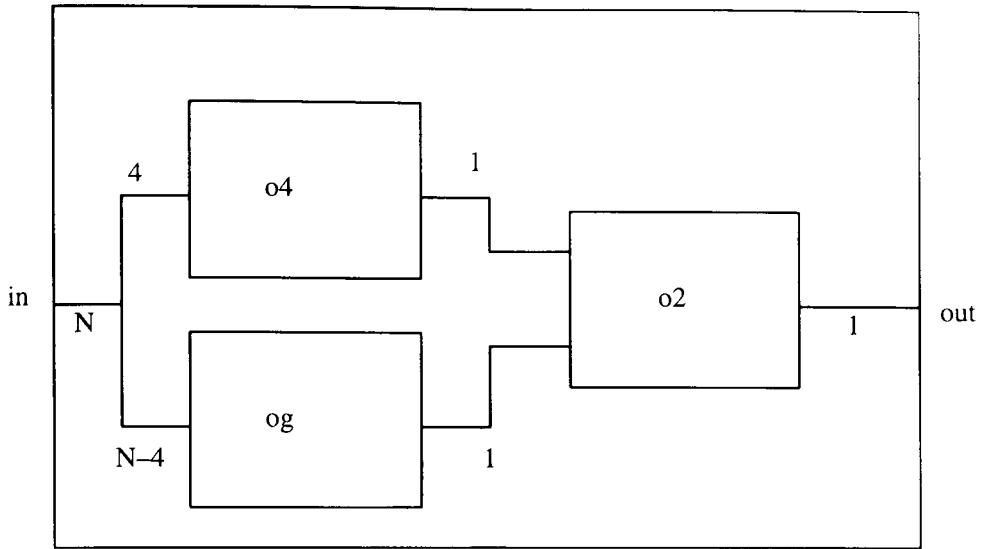


Fig. 7. Recursive structure of n -input OR gate, $n > 4$

3.9.5. n -bit register

The n -bit register is another parameterised cell, popular as a layout benchmark (the layout generated for this example is shown in Appendix A). The description shows the use of recursion to achieve iteration. If ' n ' equals 1, only the basic register cell (called 'flip_flop', not shown here) is used (lines 23–27). Otherwise, a basic cell plus an $(n-1)$ -bit register are used (lines 28–34). The example also shows the use of the 'join' function to join busses together (line 33). In this case, the output bus 'out' is formed by joining together two internal busses, the output busses of instances 'f' and 'tailreg'.

```

1/ BLOCK Register (n : integer)
2/     HAVING (in : posint(n)
3/           clock : bit) :
4/           (out : posint(n))
5/     INHERIT bit FROM hal
6/     TYPE
7/         posint(n: integer) ::=
8/         { IS [0 .. (2 ** n) - 1]
9/         REPRESENT BY nbits: WIRE [n]

```

```

10/          WITH MAPPING sigma(0, n - 1, nbits)
11/          }
12/      WHERE
13/          decode(w: WIRE): integer ::=
14/              IF (w == low) THEN
15/                  0
16/              ELSE
17/                  1
18/      INTENDED BEHAVIOUR
19/          WHENEVER change (clock) :
20/              WITHIN (10)
21/                  SET out = in;
22/      USE STRUCTURE
23/      (n == 1):
24/      { INSTANCE f: flip_flop
25/          USING f(in, clock)
26/          MAKE out ::= f.out
27/      }
28/      (n > 1):
29/      { INSTANCE f: flip_flop
30/          tailreg: Register(n-1)
31/          USING f(head(in), clock)
32/          tailreg(tail(in), clock)
33/          MAKE out ::= JOIN(posint(n) | f.out, tailreg.out)
34/      }
END

```

The n -bit register is the main example used in the chapter on the viewer. Diagrams can be found there.

In general, an implementation that recursively implements a structure of N elements produces the following kind of design hierarchy (see Fig. 8):

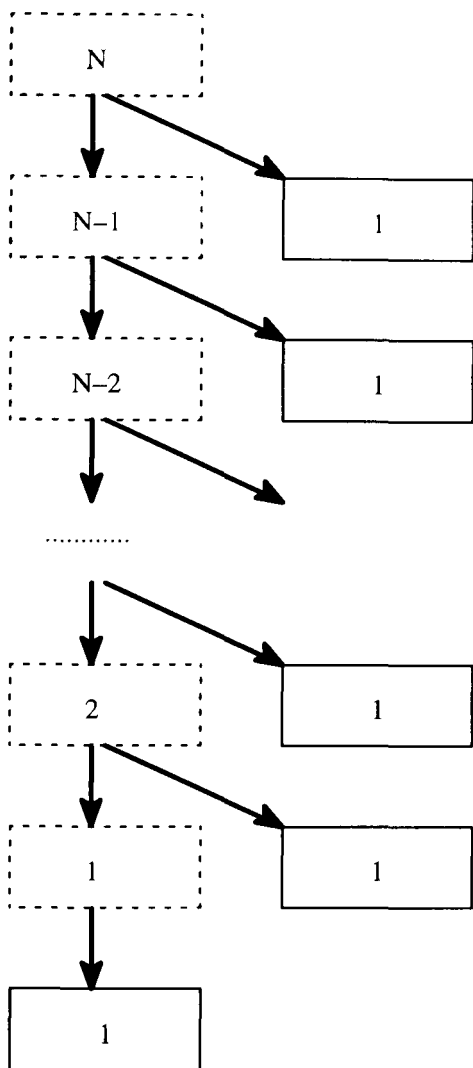


Fig. 8. Original hierarchy

In this picture, dotted boxes denote recursive calls, while the solid boxes indicate a single implementation of a basic cell. The numbers in the dotted boxes refer to the value of the actual parameter used in the recursive call.

Such a structure is ideal for the purpose of formal verification, but highly inefficient when it comes to producing layout. In this case, the structure must be flattened first. Flattening produces the following hierarchy (see Fig. 9):

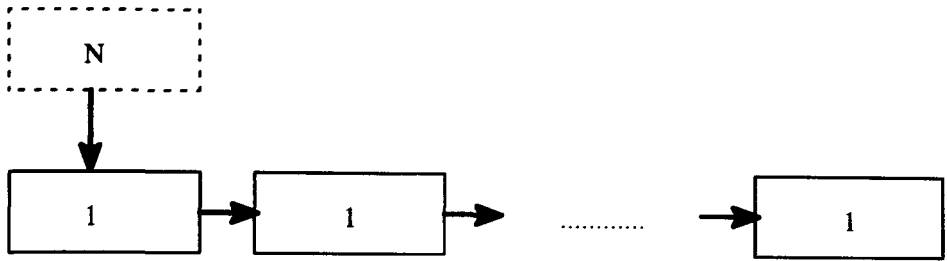


Fig. 9. Flattened hierarchy

3.9.6. Systolic Array

This example shows the use of recursion in two dimensions, to create a rectangular array of cells (frequently called a systolic array). The design consists of a square array of alternating cells called 'black' and 'white' for convenience. The design has a block called 'wr' (meaning 'white row', lines 8–37), which recursively builds a row of cells in a way similar to the n-bit register from the previous example, except that in the recursive call it uses a block called 'br' (meaning 'black row') which has a black cell at the front (line 30). The 'br' block (lines 38–67) does something similar with 'wr' (line 60). What we in fact have here, then, is daisy chain recursion whose effect is a row of alternating black and white cells. These are called from within the block 'wrs' (lines 68–96) which builds rows on top of each other in order to build a rectangular grid. This is again done recursively, in a similar manner as the 'wr' block, by calling a block called 'brs' (line 88). The example also shows the use of the **collapse** keyword (e.g. line 30) and the 'place' statement (e.g. line 31) to achieve satisfactory layout, and the use of the **wire** keyword (e.g. lines 9–12).

```

1/ BUILD

2/   { INSTANCE c: wrs(16, 16)
3/     USING c(ain, bin)
4/     MAKE aout ::= c.oute
5/         bout ::= c.outs
6/   }

7/   GIVEN
  
```



```

8/      BLOCK wr(n: integer)
9/          HAVING (inn @n: WIRE[n]
10/              inw @w: WIRE ) :
11/              (outs @s: WIRE[n]
12/              oute @e: WIRE )
13/      INHERIT
14/          whitecell FROM '$layout$test:'
15/      INTENDED BEHAVIOUR
16/          WHENEVER
17/              (rise(clock)):
18/                  WITHIN (10 * n)
19/                      SET
20/                          oute = inw;
21/      USE STRUCTURE
22/          (n == 1):
23/              { INSTANCE w: whitecell
24/                  USING w(inn, inw)
25/                  MAKE oute ::= w.oute
26/                  outs ::= w.outs
27/              }
28/          (n > 1):
29/              { INSTANCE w: whitecell
30/                  b: br(n-1, COLLAPSE)
31/                  PLACE w ; b
32/                  USING w(head(inn), inw )
33/                      b(tail(inn), w.oute)
34/                  MAKE oute ::= b.oute
35/                      outs ::= join(WIRE[n] | w.outs,b.outs)
36/              }
37/      END

```

```

38/      BLOCK br(n: integer)
39/          HAVING (inn @n: WIRE[n]
40/              inw @w: WIRE ):
41/              (outs @s: WIRE[n]
42/              oute @e: WIRE )

43/      INHERIT
44/          blackcell FROM '$layout$test:'

45/      INTENDED BEHAVIOUR
46/          WHENEVER
47/              (rise(clock)):
48/              WITHIN (10 * n)
49/              SET
50/              oute = inw;

51/      USE STRUCTURE
52/          (n == 1):
53/              { INSTANCE b: blackcell
54/                  USING b(inn, inw)
55/                  MAKE oute ::= b.oute
56/                  outs ::= b.outs
57/              }
58/          (n > 1):
59/              { INSTANCE b: blackcell
60/                  w: wr(n-1, COLLAPSE)
61/                  PLACE b ; w
62/                  USING b(head(inn), inw)
63/                      w(tail(inn), b.oute)
64/                  MAKE oute ::= w.oute
65/                  outs ::= join(WIRE[n] | b.outs,w.outs)
66/              }
67/      END

```

```

68/      BLOCK wrs(n, siz: integer)
69/          HAVING (inn @n: WIRE[siz]
70/              inw @w: WIRE[n] ):
71/              (outs @s: WIRE[siz]
72/              oute @e: WIRE[n] )

73/      INTENDED BEHAVIOUR
74/          WHENEVER
75/              (rise(clock)):
76/                  WITHIN (10 * n)
77/                      SET
78/                          outs = inn;

79/      USE STRUCTURE
80/          (n == 1)      :
81/              { INSTANCE w      : wr (siz, COLLAPSE)
82/                  USING   w (inn, inw)
83/                  MAKE
84/                      outs ::= w.outs
85/                      oute ::= w.oute
86/              }
87/          (n > 1):
88/              { INSTANCE b: brs(n-1, siz, COLLAPSE)
89/                  w: wr(siz, COLLAPSE)
90/                  PLACE b / w
91/                  USING w(b.outs, head(inw))
92/                      b(inn, tail(inw))
93/                  MAKE outs ::= w.outs
94/                      oute ::= join(WIRE[n] | w.oute,b.oute)
95/              }
96/      END

97/      BLOCK brs(n, siz: integer)
98/          HAVING (inn @n: WIRE[siz]

```

```

99/             inw @w: WIRE[n] ):
100/             (outs @s: WIRE[siz]
101/             oute @e: WIRE[n] )

102/             INTENDED BEHAVIOUR
103/             WHENEVER
104/             (rise(clock)):
105/             WITHIN (10 * n)
106/             SET
107/             outs = inn;

108/             USE STRUCTURE
109/             (n == 1):
110/             { INSTANCE b: br(siz, COLLAPSE)
111/             USING b(inn, inw)
112/             MAKE
113/             outs ::= b.outs
114/             oute ::= b.oute
115/             }
116/             (n > 1):
117/             { INSTANCE w: wrs(n-1, siz, COLLAPSE)
118/             b: br(siz, COLLAPSE)
119/             PLACE w / b
120/             USING b(w.outs, head(inw))
121/             w(inn, tail(inw))
122/             MAKE outs ::= b.outs
123/             oute ::= join(WIRE[n] |b.oute,w.oute)
124/             }
125/             END

```

3.9.7. Sigma function

The 'sigma' function is an important behavioural function: it converts a bit pattern on a bus into an (unsigned) integer. This function is of course named after the well known mathematical operator with the same name. It makes use of the 'decode' function which converts

the value on a single bit wire to 0 or 1 as appropriate. Again, recursion is used as the basic mechanism to access all the wires of the bus; if there is only one wire in the bus it is just decoded, otherwise the low order bit is decoded and 2 times the recursive call on the rest of the bus is added (line 5).

```

1/      Sigma(lower,upper:integer, w: WIRE[*]) : integer ::=
2/          IF (upper == lower) THEN
3/              Decode(w[lower-1])
4/          ELSE
5/              Decode(w[lower-1]) + 2*Sigma(lower+1,upper,w)
6/      Decode(w:WIRE):integer ::=
7/          IF (w == low) THEN
8/              0
9/          ELSE
10/             1

```

3.9.8. Error Corrector

This example of a behavioural description is in fact a complete program; it shows a non-trivial signal processing example – an error correcting function for a decoder. The decoder is taken from [42]. It takes as its input a 32 bit number, performs an error check on it, and corrects a specified set of disturbances. The specification suggests that when the function is activated, the input is stored at a specific array variable 'a[i]' (line 6) on each rising edge of a clock signal (neither 'a' nor 'i' have been declared in the example for the sake of brevity). When output is required, it is set to 'a[i]', with an error correction determined by the function 'error' (line 11). The actual program (lines 13–43) consists of a collection of nine functions, many of which are recursive.

```

1/ WHENEVER rise(clock) =>
2/     (select1 == 1):
3/         WITHIN (t1)
4/             SET
5/                 i = (i + 1) MOD 32
6/                 a[i] = input;

```

```

7/      (select2 == 1):
8/      WITHIN (t2)
9/          SET
10/              i = (i + 1) MOD 32
11/              output = a[i] + error(a[i],i);
12/      WHERE
13/      error(a: byte[32], i: integer):byte ::=
14/          IF correctable(a[i],i) THEN eval(a, 0)
15/          ELSE 0
16/      correctable(a: byte[32], i: integer):BOOLEAN ::=
17/          (eval(a, 0) == alfapow(0*i, eval(a, 0))) AND
18/          (eval(a, 1) == alfapow(1*i, eval(a, 0))) AND
19/          (eval(a, 2) == alfapow(2*i, eval(a, 0))) AND
20/          (eval(a, 3) == alfapow(3*i, eval(a, 0)))

21/      eval(a: byte[32], j: integer):byte ::=
22/          sum(31, a, j)

23/      sum(i: integer, a: byte[32], j:integer):byte ::=
24/          IF (i == 1) THEN partsum(a, i, j)
25/          ELSE exor(partsum(a, i, j), sum(i-1, a, j))

26/      partsum(a: byte[32], i, j: integer):byte ::=
27/          alfapow(j*i, a[i])

28/      alfapow(n: integer, g: byte):byte ::=
29/          IF (n == 0) THEN g
30/          ELSE alfax((alfapow(n-1, g)), 1, 8)

31/      alfax( g: byte, i, j: integer):byte ::=
32/          IF (i < 6) AND (i > 2)
33/          THEN (EXOR(g[j], g[i-1])) +

```

```

34/                                     2*(alfax(g, i+1, j)))
35/                                ELSE IF (i == 1) THEN (g[j] +
36/                                     2*(alfax(g, i+1, j)))
37/                                ELSE IF (i == j) THEN g[i-1]
38/                                ELSE g[i-1] + 2*(alfax(g, i+1, j))

39/                                exor(g, a: byte, i: integer):byte ::=
40/                                IF (i == 8) THEN EXOR(g(i), a(i))
41/                                ELSE EXOR(g[i], a[i]) + 2*(exor(g, a, i-1))

42/                                EXOR(a,b: bit): bit ::=
43/                                ((~a) AND b) OR ((~b) AND a)

```

The details of the algorithm are not of great interest here. The Transformer chapter shows how this behavioural description can be subjected to formal interactive transformations, and then converted into a correct structural implementation. This demonstrates the benefits of using high level synthesis tools.

3.9.9. Traffic Light Controller

This section describes a popular example of a finite state machine description: the Mead and Conway traffic light controller. The behaviour describes the operation of a controller of a set of traffic lights at a junction between a busy main road and a quiet farm road. The controller takes inputs from road sensors and from a timing device (line 3), and cycles the lights as appropriate (line 4), whilst giving higher priority to the main road. The example shows the use of a mapping function within the type declaration for the light colours (lines 8–18). The specification of the behaviour (lines 19–93) shows the use of a number of state variables (lines 20–21), e.g. for remembering the state of the timers. Line 27 shows that it is the clock tick which is the main event in the system. When this happens, the simulator will find the conditions (as specified on lines 29, 34, 39, 47, 52, 65, 73, 80 and 85) that are true, and will execute the statements following the condition. This generally leads to assignment of new values to state variables, and therefore to execution of new statements at the next clock tick.

```

1/ BLOCK tlc
2/   HAVING (carhere,longtimeout,shorttimeout: WIRE):
3/           (timer:WIRE
4/           road,farmroad: colour)
5/   TYPE colour ::= { IS (green, yellow, red)
6/           REPRESENT BY lcols: WIRE [2]
7/           WITH MAPPING
8/           IF (lcols[1] == low) THEN
9/           (
10/              IF (lcols[2] == low) THEN
11/              (green)
12/              ELSE
13/              red)
14/           ELSE
15/              IF (lcols[2] == low) THEN
16/              (yellow)
17/           }
18/
19/ INTENDED BEHAVIOUR
20/   STATE roadgreen, roadyellow, farmroadgreen,
21/       farmroadyellow ::= boolean
22/   INITIALLY roadgreen=true,
23/       roadyellow = false,
24/       farmroadgreen = false,
25/       farmroadyellow = false
26/   WHENEVER
27/       rise(clock) =>
28/       {
29/           ~carhere AND roadgreen:
30/           WITHIN (1)
31/           SET
32/           road=green

```



```

33/                farmroad=red;
34/    roadgreen AND ~longtimeout:
35/        WITHIN (1)
36/            SET
37/                road=green
38/                farmroad=red;
39/    roadgreen AND carhere AND longtimeout:
40/        WITHIN (1)
41/            SET
42/                road=green
43/                farmroad=red
44/                timer = 1
45/                roadgreen = false
46/                roadyellow = true;
47/    roadyellow AND ~shorttimeout:
48/        WITHIN (1)
49/            SET
50/                road=yellow
51/                farmroad=red;
52/    roadyellow AND shorttimeout:
53/        WITHIN (1)
54/            SET
55/                road=yellow
56/                farmroad=red
57/                timer=1
58/                farmroadgreen=true
59/                farmroadyellow=false;
60/    farmroadgreen AND carhere AND ~longtimeout:
61/        WITHIN (1)
62/            SET
63/                road=red
64/                farmroad=green;

```

```

65/          farmroadgreen AND ~carhere:
66/              WITHIN (1)
67/                  SET
68/                      road=red
69/                      farmroad=green
70/                      timer=1
71/                      farmroadgreen=false
72/                      farmroadyellow=true;
73/          farmroadgreen AND longtimeout:
74/              WITHIN (1)
75/                  SET
76/                      road=red
77/                      farmroad=green
78/                      farmroadyellow=true
79/                      farmroadgreen=false;
80/          farmroadyellow AND ~shorttimeout:
81/              WITHIN (1)
82/                  SET
83/                      road=red
84/                      farmroad=yellow;
85/          farmroadyellow AND shorttimeout:
86/              WITHIN (1)
87/                  SET
88/                      road=red
89/                      farmroad=yellow
90/                      roadgreen=true
91/                      farmroadyellow=false;
92/          }
93/  END

```

The layout generated by the module generator tool for this example is shown in appendix A.

4. DESIGN METHODOLOGY

4.1. Introduction

After definition of the language, we now discuss how the language should be used during the design process. Designing a circuit takes the form of a loop in which ideas are formulated in the language, design tools are applied to test the validity of the work, the results of which are then used to formulate new ideas, and so on. Historically, designers worked in a bottom-up fashion, i.e. they decided what basic building blocks were required for the design, perhaps at the register transfer level or even lower, and then simulated the design to check the correctness. Such a methodology is not really suitable for a language such as STRICT, which is intended to be used at high levels, and which is first of all intended to be used with formal verification and transformation tools.

4.2. Design Methodology

The design process is divided into two distinct stages. The first tackles the production of a top-down design definition of the required component, while the second involves the development, bottom-up, of a suitable implementation for the design. The first stage attempts to partition the design, and to ensure that realistic progress towards a solution is made using that particular partitioning. Early identification of design problems is possible using such an approach. Having completed the design, the implementation is produced by piecing together the simplest components, and then using them to construct the next level up, and so on.

This methodology therefore results in designs which are hierarchical (or tree-structured) in organisation. The external nodes of the structure (or leaf cells) identify three types of object. The first are primitives. Primitives are objects which are provided by a particular implementation of STRICT and its design environment. They are organised into libraries, and will define the basic building blocks of the design system. As primitives, they will be 'hard'. That is, if used, they will always appear the same in function or visible layout. The second form of external node are objects which are defined in the STRICT notation, and are therefore 'soft', but which have been tried and tested previously. These objects will effectively be the entry points to previously completed designs which are now located in

public libraries. These two objects, as a consequence, provide two forms of library for use by a designer. The third form of external nodes are incomplete. That is, they contain no information regarding the implementation, and just provide a behavioural description. Such nodes cannot be fabricated and require further design work to be done by the designer.

Every node (or component) in the design tree has a specification, and ultimately, an implementation. The specifications are used in validating the design as it progresses and the implementation ultimately enables the layout to be generated. An incomplete design can still be validated as a whole, since the upper level nodes 'simulate' the required function of the currently missing lower level nodes.

The STRICT designer is responsible for initially identifying the required behaviour, plus any constraints that must be satisfied, and capturing this information in the STRICT language. He is then responsible for decomposing the design into suitable sub-components and providing suitable specifications and constraints, perhaps by using appropriate high level synthesis tools. He then defines the connectivity between the upper and lower level components. Once this is completed, he is responsible for checking the validity of his decomposition, unless the decomposition was generated automatically by design tools. This may result in the need to modify the design as stated at either the upper or lower level. For example, constraints at the upper level may need to be relaxed or sub-components changed for better alternatives. Once the designer is satisfied, he may then resume the process with one of the sub-components, using its specification as the required behaviour. This process is repeated until the availability of primitives or previously defined components renders further decomposition superfluous.

4.3. Design system

In order to support the above design methodology, it is obvious that a variety of design tools are required. Such tools need to handle the validation of designs, the production of layout, the provision of feedback to the designer, the definition of new primitives, and so on. It is vital, however, that these tools are integrated. That is, it must be possible for a designer to switch easily between say simulating a portion of a design, to altering it and then simulating

it once again. This means that the tools must be designed in such a way that communication between them is straightforward.

The tools provided within the design system must be able to support partial as well as complete designs. Part of the strength of the design methodology is the ability of a designer to ask questions about the current state of a design. The answers to such questions will govern the direction in which the design should proceed. This is as true of the simulator as the layout system. A designer may wish to lay out a small piece of the design, and then store it for future use. The designer should not be forced into having to produce the layout for the entire design at the same time, unless that is really what is wanted. This use of the design tools leads quite naturally to the idea of Incremental Silicon Compilation; that is, the process of tackling self-contained parts of the design, before tackling and completing the overall design.

The editor is used to enter all designer generated information. It initially accepts STRICT descriptions, and, as mentioned before, checks them for syntactic as well as certain semantic errors, so that a design once accepted by the design environment will be known to be syntactically correct, and can therefore be processed with minimal checks. A designer is prevented from leaving syntactically incorrect and incomplete STRICT programs in the system. The alternative textual and graphical interface to the editor will allow alternative views of the same design to be shown. Such alternative views will permit a designer to get a feel for how the design is fitting together in two dimensions, as well as providing a convenient mechanism for navigating around it. At all times, however, the language is the master description.

A rudimentary form of design validation is achieved through simulation. STRICT allows the designer to specify both intended behaviour and the structure the design requires to achieve this behaviour, thus making it possible for a simulator to provide feedback on whether the intended behaviour is properly implemented by the structure.

The layout suite of tools is responsible for ultimately producing the masks for the chip in the appropriate technology. The STRICT language is seen as being technology indepen-

dent, and only at the level of the primitives do issues such as which layer of metal to use, etc. become important. As mentioned before, the designer can, through the use of editor and STRICT language, influence the placement of components to a certain extent. In the short term, this is obviously desirable, but in the longer term, it is felt that as much as possible should be left to the design tools, so that the designer is not forced to specify information which is not critical to the functional design.

The simulator is not intended to be the ultimate tool as regards the validation of designs. Simulation is not able to provide a formal proof that two descriptions of the same thing are equivalent. A similar problem exists here as in the issue of testing software. Therefore validation of a design requires that formal mathematical proof on the design is necessary. Such a proof would be performed by a theorem prover, and would show that the implementation of the final circuit was an alternative version of the initial specification given by the designer at the beginning. The theorem prover would be used in much the same way as the simulator, except that the prover would operate on the formal specification. The prover would therefore be able to validate parameterised designs.

An alternative approach to using a theorem prover for post-hoc verification of an existing design is the use of formal transformations. This approach ensures that no errors are introduced during the design process, by using transformations that are known to replace one specification by an equivalent other one. The approach allows the designer to partition and optimise the design without inadvertently changing the specification.

The designer should also be allowed to import (parts of) designs from other design systems, or transport STRICT designs to other design systems, particularly if other design systems contain superior layout and fabrication facilities.

The tool set should therefore contain:

- a syntax directed text editor
- a graphics tool
- a simulator
- a layout sub-system

- a theorem prover
- a formal transformation tool
- an EDIF interface

Let us examine each of these tools in more detail.

4.4. Syntax directed editor

The use of a syntax directed editor, i.e. a text editor with built-in knowledge of the syntax of the language being edited, should help the designer to avoid introducing syntax errors into the text of the design, thereby eliminating one class of errors from the design process.

Since a syntax directed editor will be able to build a parse tree of the text, the design system can take this parse tree as its input instead of the original text.

4.5. Simulator

In spite of the well known disadvantage of simulation (the inability to exercise large designs completely), simulators are still state of the art verification tools, although formal verification techniques are slowly making progress.

The objective of a simulator is to validate that the functional and temporal operations of the design being simulated meet their specified requirements. This will not result in a formal proof that the design is correct, but it will certainly increase confidence in its correctness.

The simulator should reflect the hierarchical design methodology which is encouraged by the STRICT language. The simulation process involves decomposing the design through the hierarchical structure defined by its linguistic description. The units of simulation are directly equivalent to the blocks of the STRICT language.

The designer, through using the simulator, should attempt to validate the intended behaviours of the blocks in the design in a top-down manner. The intended behaviour of a block in the hierarchical design tree is validated by comparing it with a simulation of the interconnected block behaviours from the next level down in the hierarchy. If the validation fails

this indicates a design error in the lower level of the hierarchy, or a need to relax the design constraints implied by the specification of intended behaviour at the higher level. When the validation has been completed, the designer may proceed down the design hierarchy validating adjacent levels in a similar manner.

The top-down approach to simulation is totally compatible with the idea of an incremental design environment; simulation is carried out as the design evolves within the design system.

4.6. Layout generator

An incremental approach to layout is compatible with a hierarchical design methodology; the natural partitions derived from the top-down design are used for the bottom-up implementation of the layout. The bottom-up activity iteratively refines the layout. The partitioning also allows libraries of predefined STRICT blocks and primitives to be used without disrupting the layout process.

Layout information should be manipulated symbolically. This allows the high level parts of the design system to be technology (production rule) independent. The final stages of physical layout for mask definition are automatic. Thus the design system is able to, and does, support multiple technologies. Furthermore, design tools such as design rule checkers will not be required since the design will be correct by construction.

4.7. Module generation

In order to make it possible to generate silicon with some efficiency, the STRICT software should incorporate a general module generator interface, and in particular a PLA generator, which can make use of certain functional specifications of components. In addition, a limited amount of interaction with the resulting layout should be achieved through the textual STRICT description, thus allowing the designer to influence the placement of components in order to optimise the layout produced.

4.8. Graphical tools

Graphical representation is a powerful method for representing designs. Graphical displays have the drawback that they cannot be used for formal manipulation, but they can give very accurate feedback to the designer, something that pages of text cannot easily do.

The design system would therefore have a tool that would allow STRICT descriptions to be viewed in graphical form, but would otherwise not allow modifications to the design to be made. Modifications are only allowed on the original STRICT description.

There are several aspects of a design that need to be represented, ideally in a form that isolates the key information to be conveyed, and suppresses other data which are not relevant to the purpose:

- The behaviour;
- The structure;
- The control flow;
- The resource usage.

The behaviour is better presented in textual form (it needs to be a concise model for simulation or a formal specification for use in a theorem prover). The others are better expressed in a graphical form because of the need for human interaction. An example is the allocation of resources. Both structure and control flow could be represented either textually or graphically, and there is a case for doing both. Present schematic diagrams usually represent the structure of a system by showing graphically the major components and their interconnection [6]. Additionally, many chip architectures use a common data path to implement several different operations. A structural diagram of the data path must therefore be complemented by a control flow diagram such as a Petri Net, which can then be used as an aid to comprehension. A graphical view of the resources used by a particular implementation would also give useful insight into the state of the design. To summarise, the aim is to provide a comprehensive set of graphical tools to support the STRICT language which captures both the behaviour and the structure of hardware designs, even though it has features like recursion, which are not easily shown in graphical form.

4.9. Formal Verification tools

A relatively new development in the area of VLSI design is the use of theorem provers to verify circuit descriptions. These require the specification of the behaviour of a hardware design in a formal mathematical manner. Such specifications can subsequently be manipulated to prove the equivalence of hierarchical or temporal properties of hardware designs, a process known as *post hoc verification*. Since (as explained before) it is claimed that functional programs can be subjected to formal verification techniques, the designer should be able to submit a proposed decomposition of a specification to a theorem prover. Successful proof of such a decomposition should give great confidence in the correctness of a design.

The design of theorem provers is a research area in its own right, and it was never the intention to write one from scratch as part of the STRICT design environment.

We therefore examined the theorem provers which were currently available, to determine how suitable they were for the verification of hardware, and whether they would fit into the STRICT environment. The prover eventually chosen was the Boyer–Moore Theorem Prover [9] for reasons explained in chapter 10.

4.10. Transformational synthesis

One of the major problems of post hoc verification is that it requires considerable effort, and a large amount of expertise. An alternative approach is the use of *formal transformations*, which allow the designer to start with a specification and perform *correctness preserving transformations* on this specification, until an efficient design has been generated. The designer can then automatically generate a final implementation, whilst being confident of its correctness, due to the correctness preserving properties of the transformations used.

There are very good reasons for wanting to do transformations. Suppose we need to implement a module that performs the calculation $out = (a+b) * (a+b)$. A direct translation would yield a design with two adders and one multiplier. However, a competent designer would see that both multiplicands are the same, and that the design could therefore be implemented using only one adder instead of two, in effect replacing the original calculation

with $c=a+b$; $out=c*c$. This is an example of a transformation that reduces silicon area without affecting circuit speed. Other transformations involve trade-offs between speed and area. For example, a module to implement the calculation $out=a*b*c*d$ could do this in parallel using three multipliers (which would be fast, but take up a lot of area), or it could be done serially with one multiplier and a register to hold intermediate result (which would use less area, but would be slower).

4.11. Interfaces with other systems

The only standard formalism currently in use for the exchange of designs between different CAD systems is EDIF. We therefore investigated whether an EDIF interface could be integrated into the STRICT system.

5. SYSTEM OVERVIEW

5.1. Introduction

This chapter presents an overview of the design system. A diagram of the system is presented in Figure 10. Boxes generally refer to software systems; solid lines indicate the flow of data between them. Dotted lines refer to software systems outside of the STRICT system, such as CADENCE and the Boyer–Moore theorem prover.

The main user interface to the system is the SAGA editor. It produces a parse tree of the STRICT description, which is written into a file.

Some of the tools in the system require the parse tree. However, the STRICT description may contain parameterised cells, which must be further processed before they can be used, for example, for layout. This is the task of the Builder module, which produces a so called Design Hierarchy Tree from the parse tree. The design hierarchy tree is also written to a file.

The system is essentially based around the SAGA editor and the builder, and uses interfaces to the parse tree and the design hierarchy tree to provide a path to the various subsystems.

5.2. Author's contribution

The following parts of the system as shown in Fig. 10 are the author's work.

The author wrote the STRICT grammar (in conjunction with M.R.McLauchlan). The author wrote a set of lexical routines needed by the grammar.

The author was solely responsible for writing the BUILDER module, which generates the design hierarchy tree from the parse tree.

In order to be able to read or write the parse tree or design hierarchy tree as required, the author developed a set of procedural interfaces. These are used to:

- take a data structure such as the design hierarchy tree, generated by the builder, and write it to a file in a suitable format.
- read a file containing a parse tree or a design hierarchy tree, and generate appropriate data structures which can be used by the various tools.

The interface to the simulator contains a compiler that converts STRICT behavioural expressions into an assembly language format that is executed by the simulator. This compiler was also written by the author.

A very detailed overview of the various parts of the code and how they fit together are presented later in this chapter. All in all, the total number of lines of code is about 20,000.

In addition, the author was mainly responsible for development of the ideas behind the Transformer subsystem and the Viewer subsystem. However, these tools were all implemented by others. All tools imported from elsewhere were of course also implemented by others.

We now briefly describe the various parts of the system.

5.3. The SAGA editor

The SAGA editor [15] is a syntax-directed text editor, i.e. it has built-in knowledge of the syntax of the input language through the presence of a table driven parser subsystem. It can therefore locate syntax errors, and it can give hints as to what kind of input is expected at every position in the input file.

Use of the editor requires the complete STRICT syntax to be made available in the appropriate format, and it also requires some rewriting of the lexical input routines that come with the SAGA system. A parser generator then produces a file with parse tables, which is compiled with the editor.

The editor offers the user general text editing facilities, as well as special commands that operate on entire syntactic units (sub-trees in the design), and it has special modes of operation which prevent the user from exiting the editing session after the introduction of syntactic errors.

When editing is completed, the editor dumps the entire parse tree to the file system, with the result that the parse tree is then ready to be picked up by the next set of tools.

The editor is described in chapter 6.

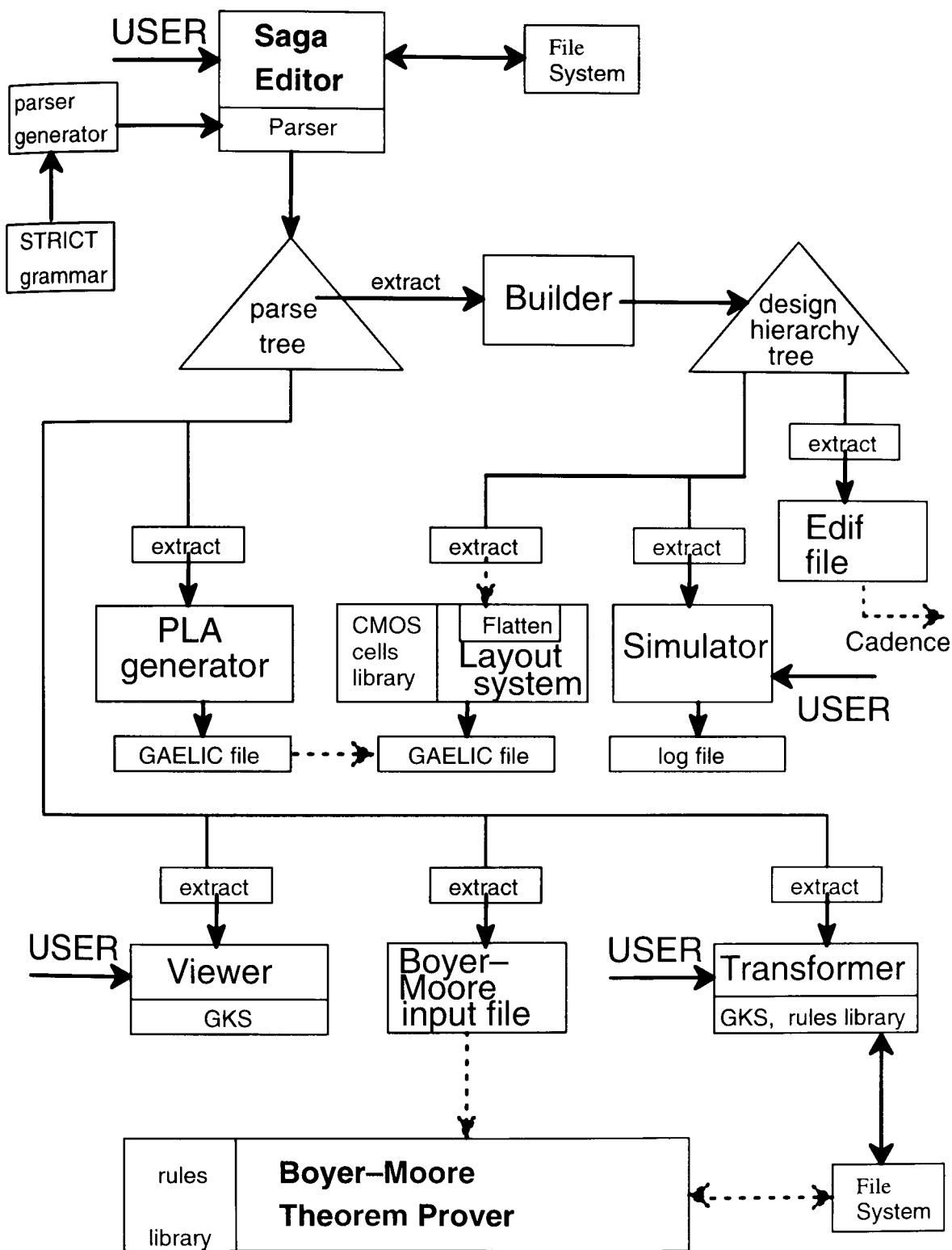


Fig. 10. Overview of the STRICT system

5.4. The BUILDER

The BUILDER module is used in cases where design tools need the actual design hierarchy. This is particularly so in the case of the layout subsystem and the simulator. For example, if the parse tree contains the parameterised description for an n -bit register, and the 'build' section asks for a 16-bit version, BUILDER will produce a *design hierarchy tree* with 16 levels, assuming that the register was specified recursively. All top level constants, as specified in the 'build' section, will permeate throughout the entire design, determining the actual sizes of busses, the depth of recursive calls, which parts of 'case' statements will be used, and so on. The similarity with the difference between formal and actual procedure parameters in programming languages is quite appropriate here.

BUILDER uses the procedural interface to the parse tree (see the section on procedural interfaces below), and it will generate a file with the complete design hierarchy tree, including modules fetched from libraries.

BUILDER is described in chapter 7.

5.5. Procedural interfaces

In order to allow various tools to access the design, a set of procedural interfaces were written for this purpose. These correspond to the boxes marked 'extract' in Figure 10.

There are two different interfaces: one to access the parse tree, and one to access the design hierarchy tree. The latter comes in two variations: one to extract the structure, and one to extract the structure and the behaviour. The second version was written specially for interfacing the simulator.

5.5.1. Parse tree interface

The parse tree access interface performs the following functions:

- it opens the parse tree file if required;
- it performs a recursive descent of the tree, and builds a data structure from it.

The data structure is then available to the calling program. The calling program can then locate the required data, using a comprehensive set of search routines which accompany the interface.

5.5.2. Design Hierarchy Tree interface – simulator

The interface takes the form of a number of procedures, which perform the following functions.

- retrieve the root instance of the tree.
- retrieve the list of block instances generated by the decomposition of a block. If the specified instance cannot be decomposed then an appropriate flag is set. This procedure can be used to extract the design hierarchy in a recursive fashion, by calling it using the instances returned by a previous call.
- retrieve a generic block record and its associated data structure, which includes the behavioural description.
- initialise the interface. A flag is set if this is successful.
- terminate the interface.

5.5.3. Design Hierarchy Tree interface – layout

One obvious difference between the simulator and the layout interfaces is that the simulator only reads data, while the layout interface both reads and writes data, for example to update routing information or add different versions of cells.

The interface contains procedures to:

- read or write the first record from the design hierarchy tree file, in order to extract or deposit technology related and other information.
- read or write complete cell descriptions, including netlists and other information.
- initialise or terminate the interface.

5.6. Tool interfaces

5.6.1. The Simulator interface

The STRICT simulator interfaces to the design hierarchy tree, and has been written to make maximum use of STRICT languages features with a view to efficiency.

The simulator interface is described in chapter 8.

5.6.2. The Layout interface

The STRICT system has two layout interfaces. The first is a GAELIC based system developed at Newcastle by Kinniment [45], which was used as a research tool into layout generation and routing algorithms. The second is an EDIF interface for input to other design systems, notably CADENCE which is available at Newcastle. Currently, only CADENCE offers a direct route to silicon. All development work on the GAELIC based system has now stopped.

The layout interfaces are described fully in chapter 8. Example outputs are shown in appendix A.

5.6.3. The Module Generator interface

Since the STRICT language has a facility for designating module generators, an extraction mechanism is available. PLA's are an obviously important part of most designs, and so an interface to a PLA generator has been provided. This is targeted towards the GAELIC based system, and uses a separate language called STATIC to achieve optimisation and generation of layout.

The module generator interface is described in chapter 8. Example output from the module generator is shown in appendix A.

5.6.4. The Viewer

The viewer sub-system allows the designer to inspect the design in graphical form, using a mouse driven window interface. It has facilities for moving through the hierarchy of the design, inspecting connections, calculating the resource usage, reducing information overload, etc. The viewer interfaces to the parse tree produced by the SAGA editor.

It is described in chapter 9.

5.6.5. The Boyer-Moore interface

In order to investigate the suitability of STRICT for the purpose of formal verification, the STRICT system has been equipped with an interface to the Boyer-Moore Theorem Prover, a widely used tool for formal verification. It interfaces to the parse tree, and produces a set of input statements for the prover, which have to be processed off-line.

Formal verification issues and the Boyer–Moore interface are discussed in chapter 10; appendix A shows some output from the prover for a simple STRICT example.

5.6.6. The Transformer interface

The transformer tool is interfaced to the parse tree, and uses Boyer–Moore rewrite rules as its underlying transformation mechanism.

The interface is described in chapter 11, and some example output of the transformer is shown in appendix A.

5.7. Code developed by the author

A detailed overview of the main modules is shown in Fig. 11. Each module shows the approximate number of lines of code in brackets.

All modules make use of the parse tree interface. The PLA generator, viewer and transformer directly use the generated data structure, as does the Boyer–Moore input file generator.

The builder produces the design hierarchy tree. It accesses leaf cells from a library of components. This library contains both layout level information and behavioural information, contained in a collection of pre–built parse trees for each component. Both kinds of information are attached to the tree, in order to make both simulation and layout possible.

It was decided to implement the layout and simulator interfaces as two separate modules, largely because there is very little overlap between them.

The layout systems need all interconnections at each level in the hierarchy to be decomposed down to the pin level. Behavioural information is not needed. In addition, the GAELIC interface performs some flattening for the sake of efficiency. The data structures used (simple linked lists) are quite straightforward and allow the required operations to be performed without any difficulty.

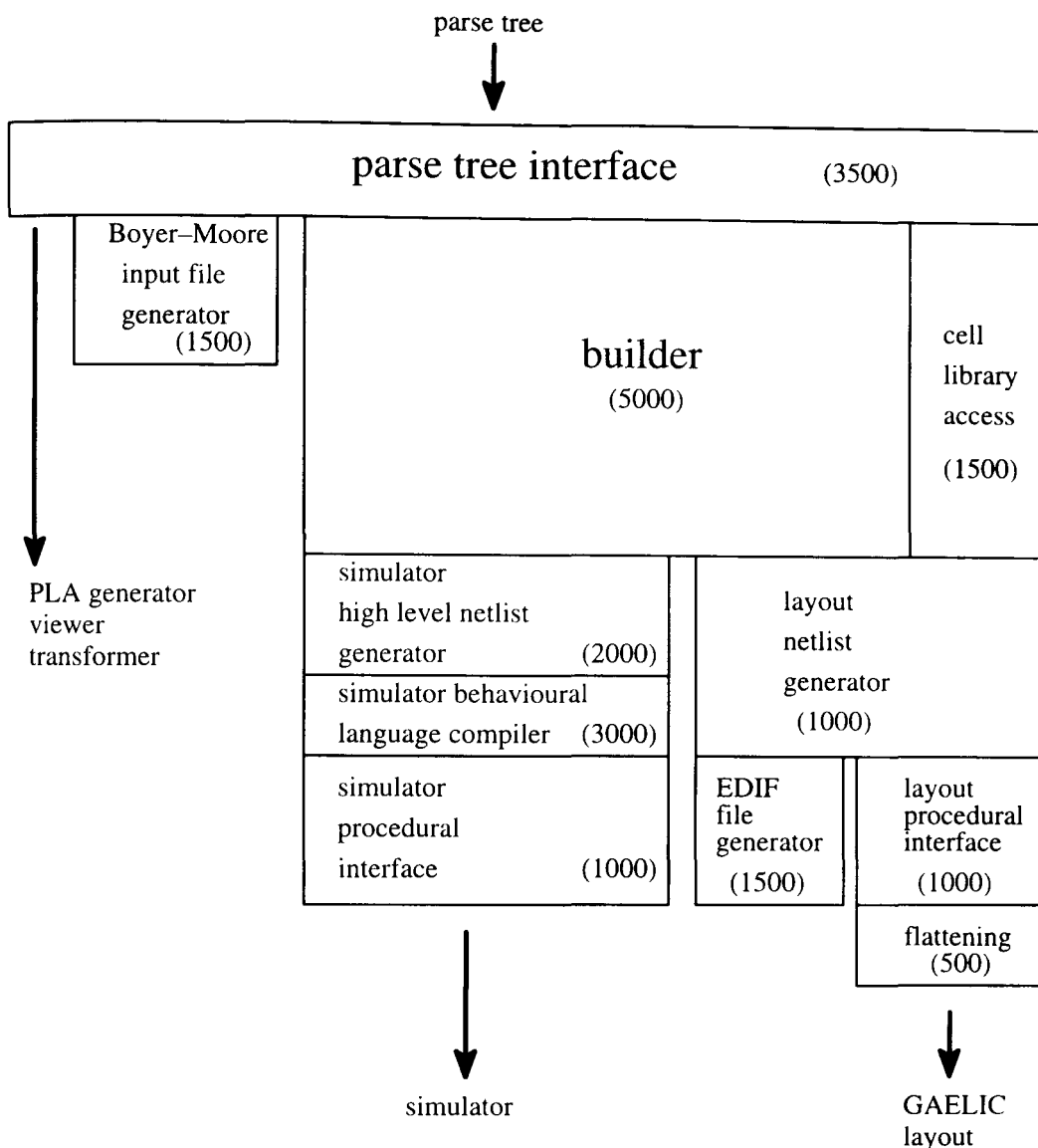


Fig. 11. Developed code

The simulator interface, however, requires all interconnections at each level in the hierarchy to be specified at the highest possible level, in order to gain the greatest possible efficiency during simulation. For example, whilst the layout interface would regard a 32-bit bus as a collection of 32 pins, the simulator will start off with regarding it as a single entity. If the bus is split, for example using the 'mostsighalf' and 'leastsighalf' functions, it will be regarded as two 16-bit busses. Since the calls to these functions may be performed in

different parts of the STRICT description, and may also be nested to arbitrary depths, much more complex data structures are needed to keep track of the number of operations performed on the busses. The advantage is that the simulator can achieve much higher efficiency when operating on these high level structures.

The simulator also requires the behavioural statements to be converted into a simple assembly language. This part of the system is a fully operational code generator, as found in compilers for high level languages.

As a result, the procedural interfaces for layout and simulation, including the file formats, are completely different.

5.8. System versions

There are in fact two versions of the design system.

The first and original one, called the SAGA system, runs under VAX/VMS, and was written in Pascal. It uses VAX GKS for screen displays. It does not contain the EDIF interface as described above.

The second version is a C translation of the first system, and runs under Unix on SUN workstations, using X-Windows as the screen interface. It uses XGKS to drive the screen display under X. It does not contain the simulator and the GAELIC layout interface, whose usefulness is now doubtful. The front-end is the second generation SAGA editor, known as Leif, which is based upon the well known GNU Emacs editor, and is written in C.

For the purpose of this thesis, we will make no distinction between these two versions since they accept entirely the same input language.

5.9. Some code statistics

We present some statistics about the size of the various subsystems, in terms of lines of code. Although the amount of code developed by the author is quite substantial, it can be seen that it forms only a small part of the overall system.

SAGA parser generator	6000
SAGA user interface	6000
SAGA editor	26000
LEIF parser generator	12000
LEIF user interface	60000
LEIF editor	19000
builder	5000
interfaces/generators	15000
viewer	13000
simulator	14000
transformer	7000
GAELIC layout system	50000
Boyer-Moore prover	90000

6. THE FRONT END

6.1. Introduction

The STRICT design system front end is a syntax directed text editor called SAGA[15]. It contains a table-driven parser, and it therefore knows the syntax of the language being edited. Apart from the usual text editing operations, it can detect when syntactic errors are made by the user.

Any text based system will need some text editing facility as its front end. Since the emphasis in the STRICT system is on correctness, it seems wholly appropriate to start at the very point where the text is typed by the user, so a syntax directed editor like SAGA is a natural choice.

The editor is screen-oriented, with most commands represented by a single keystroke. Some less frequent commands have to be typed as ordinary words. The editor operates in four different modes: view mode, select mode, modify mode, and argument mode. In view mode and modify mode the editor operates as any screen oriented text editor. In argument mode, a number of sophisticated combinations and repetitions of ordinary commands can be typed. Select mode provides tree operations, i.e. operations that work on syntactic units of the STRICT language.

SAGA is a large software system, containing a sophisticated parser generator, a window interface for VT220 terminals and a version control system, as well as the actual text editor. A number of modifications to the source code had to be made in order to make it suitable for STRICT editing, including modifications to the lexical scanning routines and addition of extra commands to call the appropriate design tools. For example, a special routine was necessary to detect the start and end of a STRICT comment.

6.2. Setting up the Editor

Before compiling the editor, it is necessary to specify a grammar for a particular language, in this case STRICT. This grammar is written in a BNF type notation, and must be suitable for LALR parsing. A parser generator will produce a number of tables from the grammar, which are included in the source code of the editor at compile time.

The STRICT grammar contains more than 300 production rules.

6.3. Overview of Operation

Once the editor has been successfully compiled, it will accept input files and allow them to be edited as ordinary text files. This includes the usual text insertion, deletion, copying, searching, replacement, etc. etc. of text. It also, however, passes all input to the parser subsystem, which uses Ghezzi's incremental parsing algorithm [28]. The parser is capable of building and modifying a parse tree for the input text. The user has the option of asking the editor for the state of the parser (e.g. are there currently any syntax errors?), and the editor is then able to tell the user where errors occur and what kind of tokens are legal at any point in the input stream. All of these features help the user to avoid and quickly rectify syntactic errors.

6.4. Editor output

Once the user exits the editor, the input file is rewritten if necessary, and also the complete parse tree is written into a separate file. If the user exits the editor whilst there are still syntactic errors present, he will be warned about this, and the parse tree will be invalidated. An invalid parse tree will be detected by subsequent tools in the design system. Some tools will actually use the memory based parse tree if possible.

Once a correct parse tree has been created, other tools in the design system will be able to use it. These tools are described in subsequent chapters.

7. THE BUILDER SYSTEM

7.1. Introduction

As explained in the chapter 5, BUILDER is used in cases where tools need the actual design hierarchy, particularly in the case of the layout subsystem and the simulator. BUILDER performs a large number of semantic checks, and generates appropriate diagnostics when required, including a dump of the design tree when an exception occurs. It also builds appropriate data structures for subsequent tools. It compiles behavioural expressions into the symbolic machine language required by the simulator.

7.2. The 'builder' procedure

7.2.1. Requirements

BUILDER is required to perform the following tasks:

- Firstly, to open the parse tree file, and check that it is valid.
- Secondly, to perform a recursive descent of the parse tree, using an appropriate part of the procedural interface, with the aim of building data structures in memory. These data structures are generally in the form of linked lists, which contain all information originally present in the text file containing the STRICT description of the design. Since a STRICT design is a collection of blocks, the top level of the data structure is a linked list which contains the names of these blocks, as well as pointers to linked lists that describe the individual properties of these blocks. The recursive descent makes use of grammar rule numbers, stored within the parse tree by the parser subsystem of the SAGA editor, to decide which right hand side alternatives of grammar rules have been applied during parsing. Since a complete parse tree is available, BUILDER is able to locate and process all type declarations within a block first, before proceeding to other parts of the block. This allows it to process block headers, which may contain typed busses, after the type information has become available.
- Thirdly, to create, after the recursive descent has been completed, a complete memory based data structure describing the entire design hierarchy tree.
- Finally, to output the design hierarchy tree in a format that is suitable for either layout or simulation. This involves generating netlists in the appropriate format, and, in the

case of simulator output, calling upon a language compiler to convert behavioural expressions into a stack language format that can be executed by the simulator.

The major part of this work, which is actually performed within the module itself, involves building the design hierarchy tree and the appropriate netlists. Building the actual design tree involves, amongst other things, replacing formal parameters with actual parameters, resolving 'if' and 'case' statements based upon the values obtained, calculating the sizes of busses and the values of function calls based upon the values of actual parameters, locating definitions of functions and blocks, and so on. All of this work has to be performed whilst descending recursively down the hierarchical levels of the design tree, and building appropriate data structures whilst doing so.

This work is done by a recursive procedure, which has the following outline:

Procedure BUILDER

- If we do not have an empty tree then locate the 'build' section,
and locate the function call that requests the top level of the design.

- Locate the block that was requested.

- Locate pointers to definitions of parameters, constants, types etc.

- Check any assertions that may be present, and quit if they fail.

- Determine sizes of all busses by expanding types.

- Find the appropriate lists of instances, by checking guards if necessary.

- If the instance is a copy of a previous cell, indicate this and return.

- For all instances in the instance list:

- allocate records, initialise, and link into the tree.

- if instances have subcells, call BUILDER recursively.

- if instances use library cells, locate and extract them.

- If layout information is required:

- build the netlist at this level.

- If simulator information is required:

- build the required netlists,

compile all behavioural expressions.

end BUILDER

The most difficult parts of this list of tasks are the determination of bus sizes and the building of netlists, both for the layout system and the simulator. A discussion of the problems follows in the next section.

As discussed above, the compilation of behavioural expressions is performed by a language compiler, which is held in a separate module. It is discussed in more detail in the next chapter.

7.2.2. Overview of problems

Let us look at a STRICT description in order to demonstrate the problems that arise during the creation of a design hierarchy tree. First, let us look at how hierarchy is created in the following (partial) example:

```
build {  
    instance f: foo(16)  
}  
  
where  
  
block foo(n: integer)  
    having (x, y: posint(n))  
    .....  
  
type posint(n: integer) ::= .....  
....  
  
structure {  
    instance a,b: bar(n-2)  
    using a(head(x), tail(y))  
    .....  
}
```

BUILDER goes through the following steps:

- It locates the function call 'foo(16)' within the 'build' section. It then searches for the definition for block 'foo', extracts its formal parameter list, and binds the name 'n' to the actual value 16.
- It calculates the sizes of busses 'x' and 'y' of 'foo'. This involves evaluating 'posint(n)'. A search for 'n' will find it as the formal parameter of 'foo', with actual value 16. The definition of 'posint' is located within 'foo', and its formal parameter, also called 'n', is now bound to the actual value of formal parameter 'n' of 'foo', i.e. 16. Note that we have different scopes for the same identifier here. Any reference to identifier 'n' within the definition of 'posint' will refer to the formal parameter 'n' of posint; if we had chosen to call the formal parameter of 'posint', say, 'm', the value of formal parameter 'n' of 'foo' would have been used instead. After the type 'posint(n)' has been evaluated, 'x' and 'y' are known to be 16 bits wide.
- A new level in the design hierarchy is created for 'foo'. BUILDER then turns its attention to the sub-modules of 'foo'.
- BUILDER finds that there are two sub-modules of type 'bar(n-2)'. It attempts to calculate the value 'n-2'. It locates 'n' as the formal parameter of 'foo' with actual value 16, and calculates 'n-2' to have the value 14. It then recursively deals with 'bar' in the same way as it did with 'foo'.
- Finally, the netlist between 'a', 'b', and 'foo' is determined, by evaluating the function calls in the 'using' section. This involves dealing with the actual parameters of the function call 'a(head(x), tail(y))', which in turn involves locating the function definitions for 'head' and 'tail' (both pre-defined), dealing with their actual parameters, and executing them.

Let us now look in more detail at the steps required to evaluate function calls. This is clearly a recursive process: the actual parameters are expressions that may involve calls to functions. As an example, let us look at the arithmetical function 'log2' discussed in the Introduction:

```

log2(100)
where
log2(n: integer): integer ::=
    if (n == 1)
    then 0
    else 1 + log2(n div 2)

```

The steps taken by BUILDER are as follows:

- The definition of 'log2' is located. There are again scope rules that must be observed; first the local 'where' section is searched, then the 'where' section associated with the behavioural specification, then the 'define' section within the block, and finally the list of predefined functions.
- The formal parameter 'n' of 'log2' is bound to value 100.
- The body of 'log2' is executed. This involves calculating the condition 'n==1' and subsequently evaluating the 'then' or 'else' branch as appropriate.
- If the 'else' branch is executed, 'log2' will be evaluated recursively, with a new actual value for formal parameter 'n'.

7.2.3. Chosen solution

In order to deal with the recursive nature of the STRICT descriptions, the BUILDER code defines a routine to deal with each structure. This routine then calls itself recursively when required.

Apart from this, it should be obvious from the description above that a lot of searching is required, determined by scope rules for various parts of the language. An efficient method for doing this is as follows. Each routine associated with each language structure has amongst its parameters a number of pointers. One of these points to the recursive structure itself. The other ones point to a set of so called *search environments* which are used to search for identifiers and their values. The environments are searched in the order in which they are passed to the routine. This order must obviously be chosen carefully, so that the scope rules are obeyed. If a routine calls itself recursively, it will normally pass the same search environments in the same order. Obviously, different language structures (such as types and

functions) will require different search environments, although in some cases they might have similar search environments which are passed in a different order.

Search environments are therefore very important. They are implemented as follows.

Each search environments is a linked list. Each structure in the list contains pairs of names and pointers to associated information. This information might for example be an integer value (if the name was defined as a constant or an actual value of a function parameter), or it might be a pointer to a data structure (for example if the name refers to a type definition or a function definition). Such structures can be concisely implemented using the PASCAL 'variant record' feature, or the C 'union' feature. As a result, a uniform search mechanism can easily be implemented.

7.2.4. Success of method

The use of recursive programming techniques appears to have had two major advantages:

- Keeping the size of the module within reasonable bounds
- Increasing the reliability of the software.

We regard BUILDER as an example of a concise and reliable software module. No significant bugs have come to light, even though the software has been extensively used by students.

Once the design hierarchy tree has been created, output is generated for the layout tools or the simulator.

7.3. Layout output

Layout output consists of a hierarchical tree description of the design, which is written into a file in a compact format. The file includes a header, which describes a number of general features such as the technology used. For each block in the tree, the following information is generated:

- names of busses, with their sizes (number of pins), types (input, output, or in-out), and the edge on which they reside;
- a list of subblocks, and perhaps their relative placement;

- a netlist, i.e. a description of the interconnections between the pins.

The file can then be processed by the placement and routing tools of the layout subsystem, before GAELIC output is finally created.

7.4. Simulator output

Simulator output also consist of a hierarchical tree description of the design, which is again written onto a file in a compact format. For each block in the tree, the following information is generated:

- names of busses, with their sizes (number of pins);
- a list of subblocks;
- a high level netlist. Unlike the layout system, simulator netlists are not decomposed down to the pin level, since the simulator can operate more efficiently on entire busses. If a 16 bit output from one block is connected to a 16 bit input bus from another block, it is not necessary to split this net up into 16 separate subnets;
- the behavioural expressions, compiled into symbolic machine language. These will be used by the simulator to schedule timing events and calculate appropriate data values.

7.5. Other subsystems

Other subsystems not mentioned above do not rely on the BUILDER, but interface directly to the parse tree. They therefore use the appropriate procedural interface to extract the required data structures.

8. INTERFACE TO TRADITIONAL TOOLS

8.1. Introduction

This chapter discusses the interfaces to what we regard as 'traditional' VLSI design tools, namely the simulator (in particular, the language compiler), the layout tools, and the module generator. These interfaces were written as part of the requirement that STRICT should not just be a high level modelling language for use with new formal verification and synthesis tools, but that the language should also be able to interface to tools that allow designers to apply traditional low level simulation techniques, and generate efficient layout at the end of the day.

8.2. Simulator

8.2.1. Extraction

The simulator uses the procedural interface on the design hierarchy tree which is generated by the builder program. The interface procedures themselves were described in chapter 5. Here we describe the data structures used in the interface.

8.2.2. Language interface

STRICT is used to define a design in a hierarchical manner. Since each level in the hierarchy must have a behavioural specification, it is possible to perform two different kinds of simulation:

- The usual kind, in which the simulator receives some input values for a decomposition at a certain level, and calculates the resulting output values. These then need to be inspected for correctness;
- So called *behavioural comparison*: first calculate the result of the top level specification, then simulate the low level decomposition, and finally compare the results, which should be identical.

The simulation model consists of interconnected, high level functional blocks. The functional blocks have behaviours which are derived from the STRICT behavioural descriptions. As a result, the interface can be divided into two areas, *generic* and *schematic*.

The schematic interface provides structural information about the block instances, their interconnections, and which generic descriptions are required. This part of the interface can

therefore be regarded as a high level netlist. The connections are not usually decomposed down to the individual pin level, but only as far as required by the STRICT description. We will not describe this part of the interface in any detail.

The generic interface provides behavioural descriptions of the simulation blocks, and their ports. This is done by converting the behavioural specification section into an appropriate data structure, and converting the actual behavioural expressions into assembly language instructions which simulate a stack machine. These instructions are then interpreted by the simulator. This stack language is now described in some detail.

8.2.3. The stack language

Behavioural descriptions must be passed across the interface between STRICT designs and the simulator. An important part of the operation of the simulator consists of evaluating the behaviour of functional blocks within the design hierarchy. The interface provides the simulator with a behavioural description of these blocks, in terms of lists of stack language operators and operands.

The operators are composed into expressions which are associated with the various STRICT clauses. There are four major data types: integer, real, boolean and bit-string. The integer, real and boolean operators do not require a detailed description here. Bit strings, which are associated with signals and ports, may be of variable length. Operators which address a pair of bit strings left pad the shorter string with zeros before processing them. Leading zeros are stripped from the resulting string before entry onto the stack.

A short notation used to describe the stacked data items in the list is given below. A data item is shown as a single letter, indicating the type of the item, and a number. The number uniquely identifies the item. The letter can be either 'i' (integer), 'r' (real), 's' (bit string), or 'd' (any type). The notation 't(s1)' means the start position within string 's1', while 'l(s1)' means the length of 's1'.

The list of instructions is as follows.

operator operands old stack new stack purpose

LOADS	port name		d1	Load signal data to stack
STORS	port name	d1		Store signal data from stack
LOADV	var name		d1	Load a state var
STORV	var name	var dat		Store state var
CALLS	funct no	depends		Call standard function
MARK				Mark parameter list
CALLU	funct name	depends		Call user function
LOADP	number		d1	Load param
RETU			d1	Return from user function
LOADC	data item		d1	Load constant onto the stack
CEQ		d1,d2	d2=d1	Compare: equal?
CNE		d1,d2	d2<>d1	Compare: not equal?
CLT		d1,d2	d2<d1	Compare: less than?
CGT		d1,d2	d2>d1	Compare: greater than?
CLE		d1,d2	d2<=d1	Compare: less than or equal?
CGE		d1,d2	d2>=d1	Compare: greater than or equal?
ANDB		b1,b2	b1 AND b2	AND booleans
ORB		b1,b2	b1 OR b2	OR booleans
EXORB		b1,b2	b1 XOR b2	Exclusive OR booleans
NOTB		b1	NOT b1	NOT boolean
LABL	label num			Define label
JMPT	label num	b1		Jump forward on TRUE
JMPF	label num	b1		Jump forward on FALSE
JMPA	label num			Jump forward always
ADDI		i1,i2	i2+i1	Add integers
SUBI		i1,i2	i2-i1	Subtract integers
MULTI		i1,i2	i2*i1	Multiply integers
DIVI		i1,i2	i2 DIV i1	Divide integers
MODI		i1,i2	i2 MOD i1	Modulus
NEGI		i1	-i1	Negate integer
EXPI		i1,i2	i2^i1	Power of an integer
ADDR		r1,r2	r2+r1	Add reals
SUBR		r1,r2	r2-r1	Subtract reals
MULTR		r1,r2	r2*r1	Multiply reals
DIVR		r1,r2	r2/r1	Divide reals
NEGR		r1	-r1	Negate real
EXPR		r1,r2	r2^r1	Power of a real
ANDS		s1,s2	s1 AND s2	AND bit strings
ORS		s1,s2	s1 OR s2	OR bit strings
NOTS		s1,(s2)	NOT s2	NOT bit string s1 and pad
EXORS		s1,s2	s1 XOR s2	EXOR bit strings

SUBS		s1,t(s2),l(s2)	s2	substring from bit string
CONS		s1,l(s1),s2	s(s2+s1)	Concatenate bit strings
RLSS		s1,shift	s2	Right logical shift
LLSS		s1,shift	s2	Left logical shift
CBS		b1	s1	Convert boolean to bit string
CSB		s1	b1	Convert low order to boolean
CSIS		is1	s1	Convert signed int to bit string
CSSI		s1	is1	Convert bit string to signed int
CUIS		iu1	s1	Convert unsigned int to bit string
CSUI		s1	iu1	Convert bit string to unsigned int

The list shows a typical set of arithmetical, logical, comparison, conversion, and flow control operations. Most instructions can only handle data items of one particular type. The conversion operators are needed particularly when operations are performed on bit strings associated with ports; these usually need converting to integers or booleans before it is possible to proceed.

We now describe how user functions are supported. The relevant operators are:

- **MARK.** This marks the bottom of a stack frame. Input parameters are loaded onto the stack after the MARK but before the function is called.
- **CALLU <name>.** Evaluate the function <name>, where <name> is a character string that is unique within the behaviour description of the simulation function block.
- **LOADP <number>.** Retrieve a parameter and pop it onto the stack. <number> indicates the position of the input parameter in the stack, up from the MARK.
- **RETU.** Return to calling routine. The function's output value is assumed to be on the top of the function's stack frame. The stack is restored to its pre-MARK state, but with the function output value on the top.

For example, the function call 'max(a, b)' with the STRICT definition of 'max' as follows:

```
max(x, y: integer): integer ::=
    if (x > y) then x else y
```

will be expressed in the calling code as:

MARK

LOADS a

CSUI

LOADS b

CSUI

CALLU max

Where the 'max' routine is compiled as follows:

LOADP 1

LOADP 2

CGTI

JMPF 99

LOADP 1

RETU

LABL 99

LOADP 2

RETU

The calling code loads the parameters and converts them to integers, before calling the routine. The routine itself loads the parameters on the stack, performs a comparison, and then performs a conditional jump, which results in the biggest of both parameters to be loaded onto the stack before the routine returns to the calling program. Note that the functional notation used in STRICT does not require the use of backward jumps, which would have made the interface data structures more complex, and the simulator algorithms less efficient and thus slower.

The 'calls' operator is used to execute predefined functions. The following special functions are currently available:

- **CHANGE.** This indicates whether a port's signal data has changed during the current simulation cycle. The input stack must contain the port's name. After execution, the output stack contains a boolean data item: **true** if changed, **false** otherwise.

- **SIZE.** This function returns the binary width of a port, i.e. the number of bits or wires in the net connected to the port. The input stack contains the port's name. After execution, the output stack contains an integer data item: the port's width.

8.3. Layout interface

The STRICT system has access to a suite of locally developed automatic floorplanning and routing tools for physical layout. These consist of a library of CMOS cells including gates, flip-flops and input/output pads; a chip planner which constructs a hierarchical floorplan; and a two layer metal router which puts in the detailed connections. These tools are fully automatic. The floorplanner works by modifying the STRICT design hierarchy to be more suitable for layout. It may flatten some areas, or partition others, the aim being to create a hierarchy of cells with similar numbers of connected subcells. Subcells are then collected together in groups, either horizontally or vertically, or in combinations of the two, so as to fit the required cell area with the smallest overlap. When the floorplan is complete, the router establishes a graph representing the channels between cells, and uses a greedy channel router to determine how wide each channel needs to be in order to contain the connections. This is an iterative process; first the horizontal channels are expanded, and then the vertical channel routing is done. Vertical channel routing may require alteration of the horizontal routing, and so on. Iterations continue until the layout is complete. The final product is a GAELIC file.

8.3.1. GAELIC interface

All necessary data is extracted from the design hierarchy tree created by the BUILDER program, using the appropriate procedural interface.

After extraction, it is often necessary to flatten the design. Flattening is a form of optimisation. Due to the frequent use of recursion in STRICT, designs will have many levels of hierarchy, which may not result in efficient layout. Every recursive function call will create a new level in the hierarchy, and each new level will have its own bounding box (causing waste of silicon area, and less efficient routing). Flattening was discussed in section 3.9.5.

Another problem in extraction is the fact that GAELIC allows only 6-character identifier names, so the layout interface will map names wherever appropriate. Again, this means avoiding name clashes.

GAELIC is a low level layout language similar to CIF [51]. It is used to define geometric shapes, and has support for defining hierarchy, scaling and rotating of shapes. None of the STRICT software directly generates GAELIC statements.

8.3.2. EDIF interface

EDIF (Electronic Design Interchange Format) [24] was developed to facilitate the interchange of data between CAD systems, possibly of different vendors, and has rapidly become the most popular format for this purpose.

EDIF is S-expression based, and allows the capturing of different views of a design, including that of a netlist.

The CADENCE VLSI Design system, a commercial package obtained through Eurochip, and with a route through to silicon, comes with EDIF input and output, and so a basic EDIF interface was written for STRICT.

Extraction uses the same procedural interface as the GAELIC interface. Generation of EDIF statements is done bottom-up, since blocks cannot be used before they have been declared.

8.4. Module generator interface

Module generators are an important part of any VLSI design system. They allow efficient design of possibly large subsystems of a design, and therefore contribute significantly to keeping the overall design cost down.

STRICT has a language construct to allow the designer to designate a particular module generator for the design of a structure. This is done through a variation of the 'use structure' statement, which allows a string to be specified between the words 'use' and 'structure'. Only one module generator is currently implemented: the PLA generator; this generator can therefore be called by writing **use 'PLA' structure**. Other generators can easily be added to the system.

A module generator uses the procedural interface to the parse tree to extract an internal data structure from the design. This data structure is then subsequently scanned for the presence of the required 'structure' statement, followed by an extraction of all relevant information.

We now describe the operation of the PLA generator.

8.4.1. PLA generation

Since the GAELIC layout system already has a PLA generator, including appropriate minimisation tools, the STRICT description of the PLA is simply translated into the input language for this system, called STATIC. Further operation is then carried out off-line.

The programming language STATIC can be used to describe the state transitions for a finite state machine (FSM). The aim of the PLA generator is to take a STRICT description and convert it into an equivalent STATIC program which, when used in conjunction with other existing programs, can be used to generate the GAELIC layout description of a PLA.

An example is shown below.

The program is called from within the STRICT Editor, running automatically when the command to build layout is entered. As described above, it is called only when the STRICT program contains the call to the PLA generator. For each FSM in the description a STATIC program is generated and placed into its own file. Each file is then processed in the background while the STRICT editing session progresses.

9. THE VIEWER

9.1. Introduction

Conventionally, hardware design has been achieved by the use of a series of schematic diagrams. These diagrams are well established [6], and a convenient way to express low level descriptions. However, a language such as STRICT is intended for the design of higher level and more complex systems. One of the problems is that STRICT has features such as recursion, which are not easily shown in graphical form.

Whilst some high level tools already exist [22], the design process, especially at high levels, is still a manual task relying on the skill and experience of the designer to trade off the extra hardware needed for most parallel implementations of an algorithm against the time required for a serial implementation. This makes it very important for a designer to be able to see the resources used by a particular implementation and how the implementation might be altered to fit with constraints of speed, silicon area or complexity. Here an appropriate representation medium can significantly reduce the design time if key aspects of the design data are provided clearly. Textual descriptions rarely express the concepts of control and data flow which may be necessary to give the designer the insight he requires into the optimisation of his algorithm and its implementation, and it is useful to illustrate the resources used during operation of the hardware by showing the total time taken and the silicon area used.

The Viewer is a high level tool intended to capture and display such properties of STRICT designs. In particular, we wish to represent the following:

- parameterisation, where present;
- recursive properties of cells;
- structure, including hierarchy, port information and netlists;
- control flow, where specified;
- resource usage, including area and speed.

It is important to realise that the aim here is to represent all properties, particularly the structural ones, in a form which makes them very clear to the user. There need be no correspon-

dence whatsoever with the eventual layout generated by automated routing tools. The display should be purely symbolic, and should be optimised for clarity towards the human user.

9.2. Author's contribution

The author was responsible for the ideas behind this tool, and for the parse tree interface that it uses. Programming of the viewer was done by a Research Associate, under the direction of the author. The ideas are presented here in the form of screen dumps that show the output of the viewer when used on the recursive n -bit register.

9.3. Extraction

The viewer uses the procedural parse tree interface to build an internal data structure from the design. This data structure is subsequently used to extract all relevant information.

9.4. The example

The description of the n -bit register presented in chapter 3 illustrates the ideas behind and the use of the viewer. In this example the input, 'in', of the register is of type 'posint(n)', an n -bit positive integer, and the intended behaviour is that the 'out' port of the register is set to the value of the 'in' port 10 time units after a clock change. The structure of the register is recursive. An n -bit version is composed of an instance of an $(n-1)$ -bit version with a flip flop:

```
BLOCK Register (n : integer)
    HAVING (in : posint (n)
           clock : binary) :
        (out : posint (n))

    INTENDED BEHAVIOUR
        WHENEVER change (clock) :
            WITHIN (10)
                SET out = in;

    USE STRUCTURE
        (n == 1) :
```

```

{ INSTANCE f : flip_flop
  USING f(in, clock)
  MAKE out :: = f.out
}

(n > 1)
{ INSTANCE f : flip_flop
  tailreg : Register (n-1)
  USING f(head(in) , clock)
  tailreg (tail(in), clock)
  MAKE out ::= JOIN(posint(n) | f.out, tailreg.out)
}

END

```

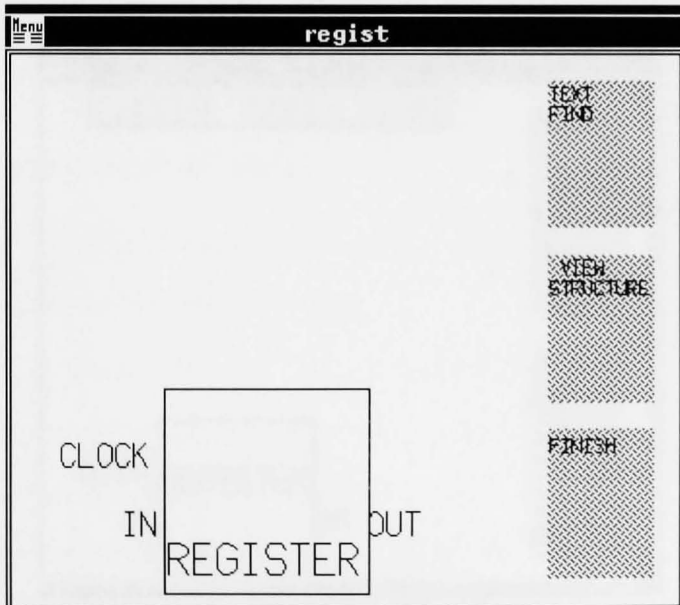


Fig. 12. Top level view

Such structures cannot normally be displayed in schematic form. In general, instances are represented as rectangles, and connections are represented by lines joining the rectangles.

When the viewer is entered, the top level of the hierarchy is shown in a window on the screen. Figure 12 shows the register as it would appear initially on the screen.

The top level view shows a named box representing the block with its ports. Input ports are shown on the left and output ports on the right. Ports which can be used both as inputs and outputs are shown with the inputs.

9.5. Traversing the hierarchy

A menu on the right hand side of the screen shows the options available to the user. At the top level of the design hierarchy the user can choose to view the block structure, and at lower levels options are available to descend further, or ascend one level. For the register example, the structure takes the form of a case statement, because the register is described recursively, and the user must choose which condition to view. Having chosen to view the structure, the prompt 'choose condition' and a menu of the conditions will be displayed (see Figure 13).

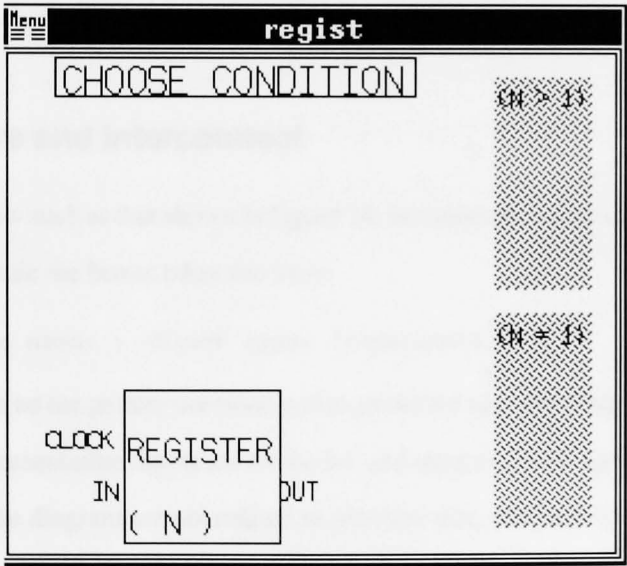


Fig. 13. View Structure option

The user picks a condition, and the structure for that condition is displayed. If the user later wants to see the structure for another condition, it is necessary to ascend to the hierarchical

level at which the choice was made and select 'view structure' (or 'view subcell') again to obtain the appropriate condition choices. Figure 14 shows the case when condition $N > 1$ is chosen.

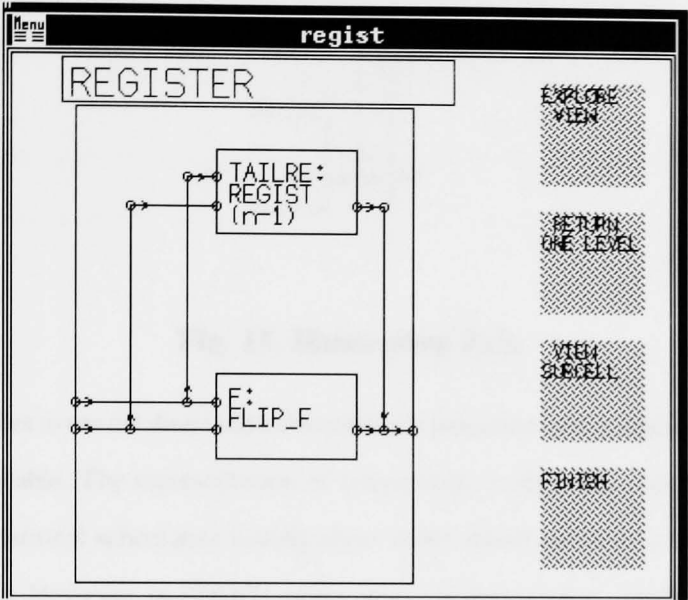


Fig. 14. Option $N > 1$ selected

9.6. Structure and interconnect

In a structural view such as that shown in Figure 14, instances are represented by boxes with text. The text inside the boxes takes the form:

<instance name> : <block type> [(<parameter>)]

All names shown on the picture are truncated to avoid the use of different font sizes. As in the top level, representation inputs are on the left and outputs on the right. The dimensions of the boxes in the diagram are not related to physical size, but have been determined for viewing purposes: all boxes in a view have the same length, and the height depends on the number of ports in an instance.

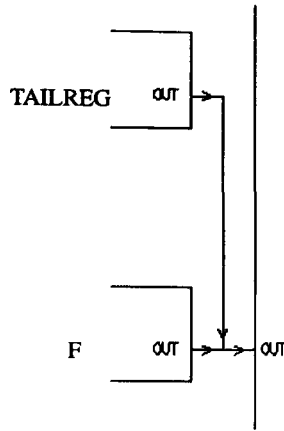


Fig. 15. Illustrating Join

If the user wishes to see the direct representation of a particular interconnection, a 'net info' facility is available. The representation of interconnect is the most complex part of the viewer. Conventional schematics usually show interconnect much as it will occur in the physical layout. However, in STRICT, interconnect is described at a higher level, and the aim of the viewer is to keep a one to one correspondence with the text. Interconnect is defined in the 'using' and 'make' sections. Each statement in the 'make' section consists of the name of an output port for the block and all the signals that connect to this port. Each statement is considered by the viewer to be an entity, that is, a net. In the 'make' statement represented in Figure 15, the wire from 'f.out' joins the bus of type 'posint(n-1)' from 'tailreg.out' to form a bus of type 'posint(n)' at port 'out'. The viewer represents this statement by one net. Each statement in the 'using' section is more complex. There is one statement for each sub block instance. Within a statement there is an entry for each input port for that instance. An entry identifies the signals connected to each input port, and each port entry is considered as a net. In the two statements shown in Figures 16, 17, and 18, the n-bit bus entering the register at port 'in' splits so that the first bit, 'head(in)', goes to the 'in' port of instance 'f' and the remaining bits, 'tail(in)', go to the in port of instance 'tailreg'. The viewer represents this by two nets.

Figure 16 shows the situation where the net representing inputs to the 'in' port of instance 'f' has been selected.

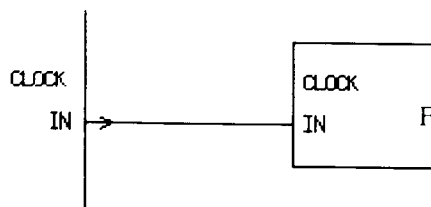


Fig. 16. Illustrating split, first part

Figure 17 shows the situation where the net representing inputs to the 'in' port of instance 'tailreg' has been selected.

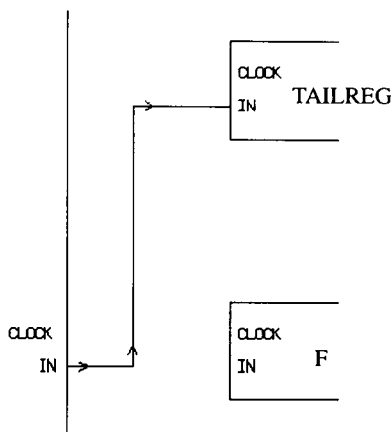


Fig. 17. Illustrating split, second part

In Figure 18, the two nets that use the 'in' signal are seen superimposed in the default view of the structure.

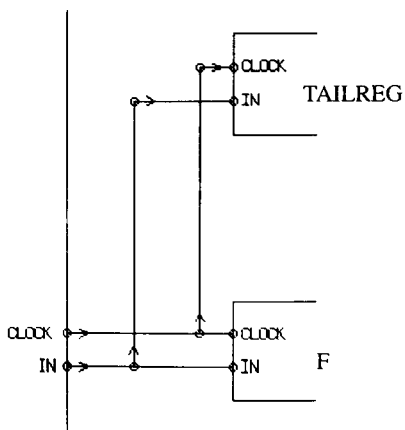


Fig. 18. Illustrating split, combined

Thus in the STRICT description, the joining of signals is described explicitly and the viewer draws the joined signals as one net. Branching of signals is represented by the superposition of the lines representing the nets at the common port as in Figure 15. Some nets may only use parts of a bus from a given port. No distinction is made between using the whole bus or only a part in the diagram, in order to keep the diagram simple.

The instances in the diagram have been arranged so that, wherever possible, data flows from left to right. When feedback loops exist signals will flow right to left. This will be indicated by the arrows on the interconnect lines.

9.7. Subcells

Display of the structure of the parent block of a given instance is done by selecting the relevant instance. If there is only one instance in the current hierarchical level, then this instance is automatically selected. If there are more than one, the user is prompted to select one.

The information displayed on entry to a particular point in the hierarchy has been deliberately restricted for clarity, but a number of options are provided to the user to allow a fuller exploration of the design at any level. These are now described.

9.7.1. Windowing

Having chosen the option 'window in', the user must select the bottom left corner and the top right corner of the area to be viewed. The selected area will be displayed as large as possible, as shown in Figure 19, keeping the proportions the same.

9.7.2. Instance information

The option 'inst info' allows the user to see the instance name, block name, and parameters displayed in full for a selected instance. After selection the information is presented in a note section at the top of the screen.

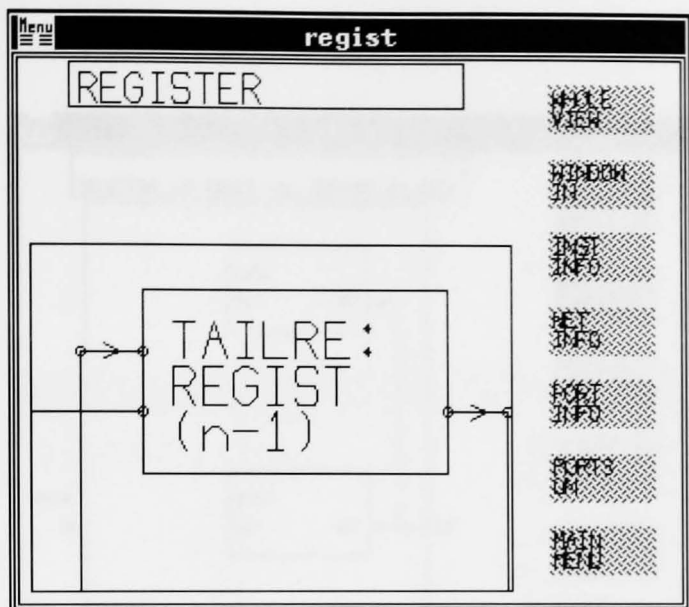


Fig. 19. A window on an area

9.7.3. Port information

Choosing the option 'ports on' causes the names of the ports, truncated to six characters, to be displayed on the diagram and, in order not to clutter the diagram, instance names are no longer displayed (see Figure 20).

Choosing the option 'port info' gives further data concerning a selected port. The information is displayed in the note section in the following form:

```
<port name> : <port type>
```

for example

```
out: posint(n)
```

9.7.4. Net information

This option allows the user to select a single net to view. As the cursor is moved over the diagram sections of nets become highlighted. The desired net can be selected and the view redrawn with only the selected net shown (see Figure 20).

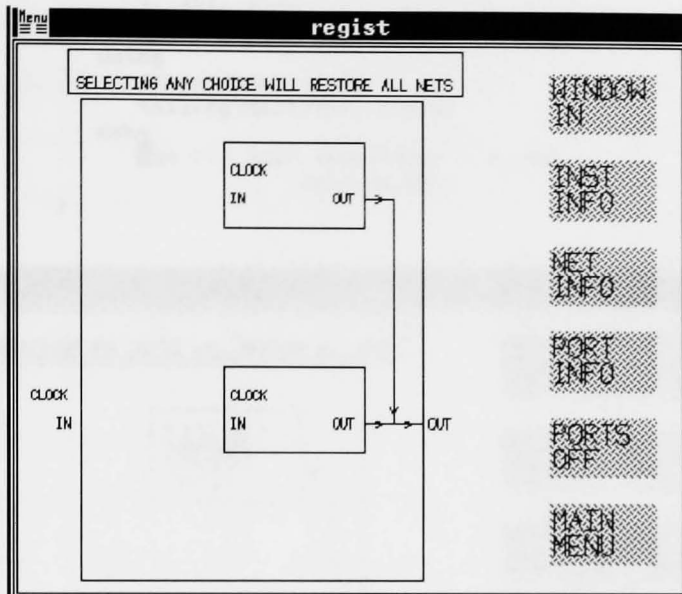


Fig. 20. Net information

9.8. Dual text–graphics representation

There are advantages in presenting a design both textually, and graphically, and the user should not be restricted solely to one or the other for either design capture or review. However, the viewer does not permit modification of STRICT designs via the graphical route, only location of data in the text from the graphics. In other words, the text representation is the master description at all times.

On selecting dual text/graphics mode, the window showing the graphics shrinks so that a window displaying the text becomes visible. When 'view structure /subcell' is selected, the block header for the block whose structure is to be viewed is found in the text (the cursor is positioned at the start of the appropriate statement).

```

(n > 1) :
{
  instance
    f: flip_flop
    tailreg: register(n - 1)
  using
    f(head(in), clock)
    tailreg(tail(in), clock)
  make
    out ::= join (posint(n) | f.out,
                  tailreg.out)
}
end

```

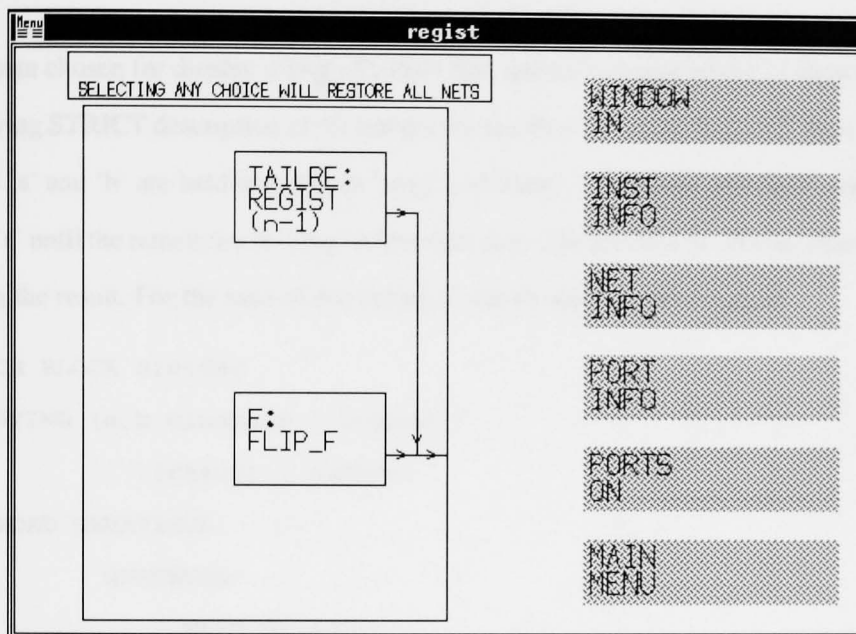


Fig. 21. Find text

If the structure is conditional on selecting the condition, the case statement for that condition is found. When a net is selected using 'net info', the appropriate statement is found (see Figure 21). When an instance is selected using 'inst info', the appropriate entry in the instance list is found. When a port is selected using 'port info', the appropriate port entry in the block header is found, if there is a block header, or if the cell comes from a library the appropriate inherit statement is found.

9.9. Control synthesis

As discussed before, in some cases the structure of the data path is insufficient to describe the behaviour of the system without an explicit view of the function of the control. In STRICT, a mechanism has been provided to capture the control flow. This allows the synthesis of a range of controller architectures such as PLA's, microprogrammed memory or asynchronous architectures, but it also allows display of the control flow in a graphical form.

The form chosen for display is that of a Petri Net, and an example of this is shown by the following STRICT description of the integer divider first shown in chapter 3. Here two integers 'a' and 'b' are held in registers 'areg' and 'breg'. 'b' is then repeatedly subtracted from 'a' until the remainder in 'areg' is less than zero. The number of subtractions less one is then the result. For the sake of convenience, the example is shown again:

```
ASYNCH BLOCK divider
  HAVING (a,b,minusone : number):
    (result : number)
INTENDED BEHAVIOUR
  WHENEVER
    WAIT divider:
      WITHIN (10)
        SET result = a DIV b
        SIGNAL divider;
USE STRUCTURE
{
  INSTANCE breg: reg
    areg: muxreg
    sub: subtract
    test : greaterthanzero
    count: counter
  CONTROL
    WAIT divider:
```



```

        SIGNAL breg, areg, count, areg.selin1
        SET count.load = 1
            count.incr = 0
    WAIT breg, areg, count, areg.selin1:
        SIGNAL test
        SET areg.selin1 = 0
    WAIT areg, count, areg.selin2:
        SIGNAL test
        SET areg.selin2 = 0
    WAIT test:
        SET ack = ~test.output,
            sub.req = test.output
    WAIT sub:
        SIGNAL areg, count, areg.selin2
        SET count.load = 0
            count.incr = 1

    USING areg(a, sub.s)
        test(areg)
        sub(areg, breg)
        breg(b)
        count(minusone)

    MAKE result ::= count
}

END

```

A block with a control section, which can be identified by the keyword **asynch** which prefixes **block**, is assumed to have in addition to its explicit inputs and outputs, two control lines. All ports involved in control handshaking are indicated by arrows by the viewer.

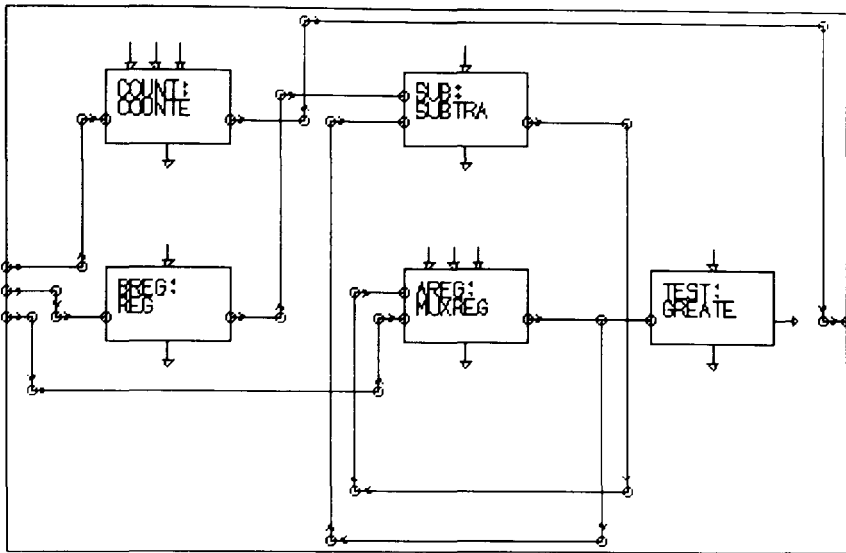


Fig. 22. Divider structure

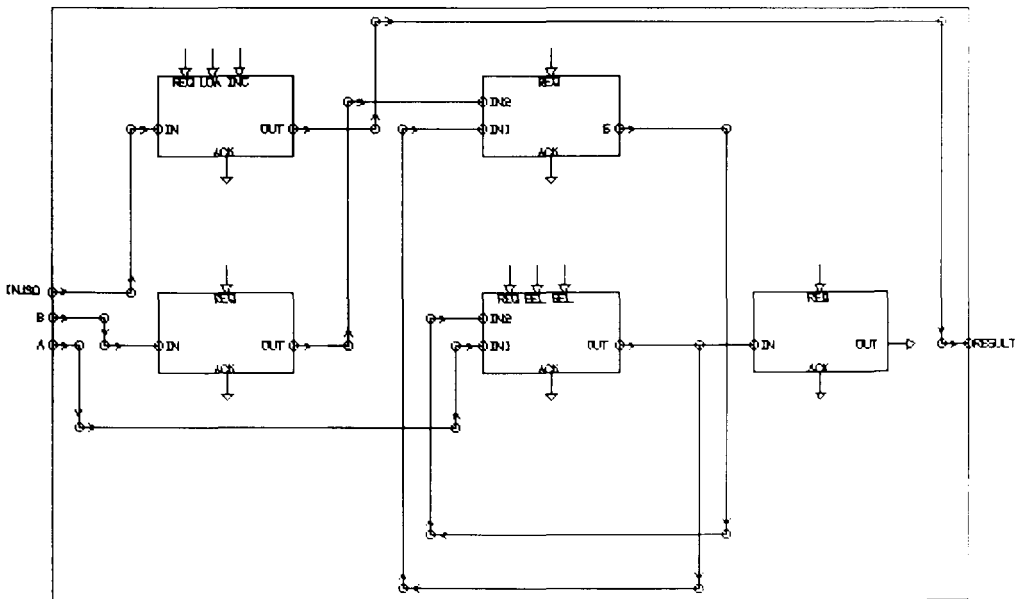


Fig. 23. Divider structure – pin names

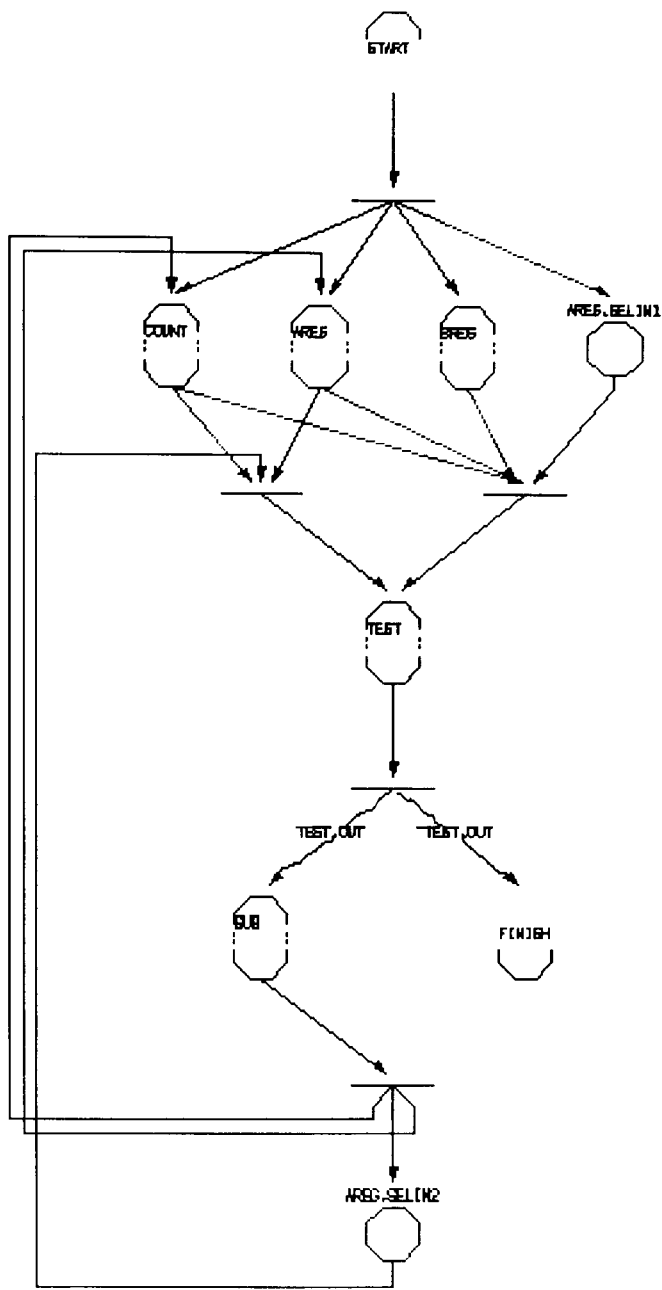


Fig. 24. Control flow Petri Net

The structure of the divider, produced from the STRICT text, is illustrated in Figures 22 (which shows the instance names) and 23 (which shows the pin names), whilst the control flow is shown in Figure 24. Since there is a one to one correspondence between statements

in the 'control' section of the language, and the resulting Petri Net form of the control diagram, both representation and synthesis of the control hardware are straightforward. In Figure 24 each 'wait' statement corresponds to a transition in the Petri Net, and each 'signal' variable represents a place in the net which receives a token when the preceding transition is fired. Places also provide the means by which the operation of a component in the structural diagram is invoked, and signalling between the control graph and the data path, or structural graph is shown by arrows in or out of the components.

9.10. Resource allocation

The development of efficient automatic physical design tools has placed an increasing emphasis on the need for tools to assist in the synthesis process, by providing the means by which insights into the nature of an algorithm and its implementation can be gained.

The major constraints on a hardware system are the function that must be implemented together with the resources that must be deployed in order to achieve that implementation. In some cases it is important to compute the required function in the minimum time, and in others the most significant factor is that the silicon area must fit within a predefined limit. Often there is a possible trade off between the time and area used, but it is always necessary for the designer to be aware of the resources required by different algorithms and implementations as well as being able to optimise a particular implementation for either time or area.

STRICT captures both the structure and the detailed timing of an implementation in the structural and behavioural sections respectively. Given that an annotation mechanism also exists for indicating the silicon area of each block, it is possible to show the relationship between area and time, using a Gantt Chart [19], in a way which aids understanding of the resource usage. The assumption here is that an implementation has already been done in which the scheduling of operations on to blocks is explicit in the control flow description.

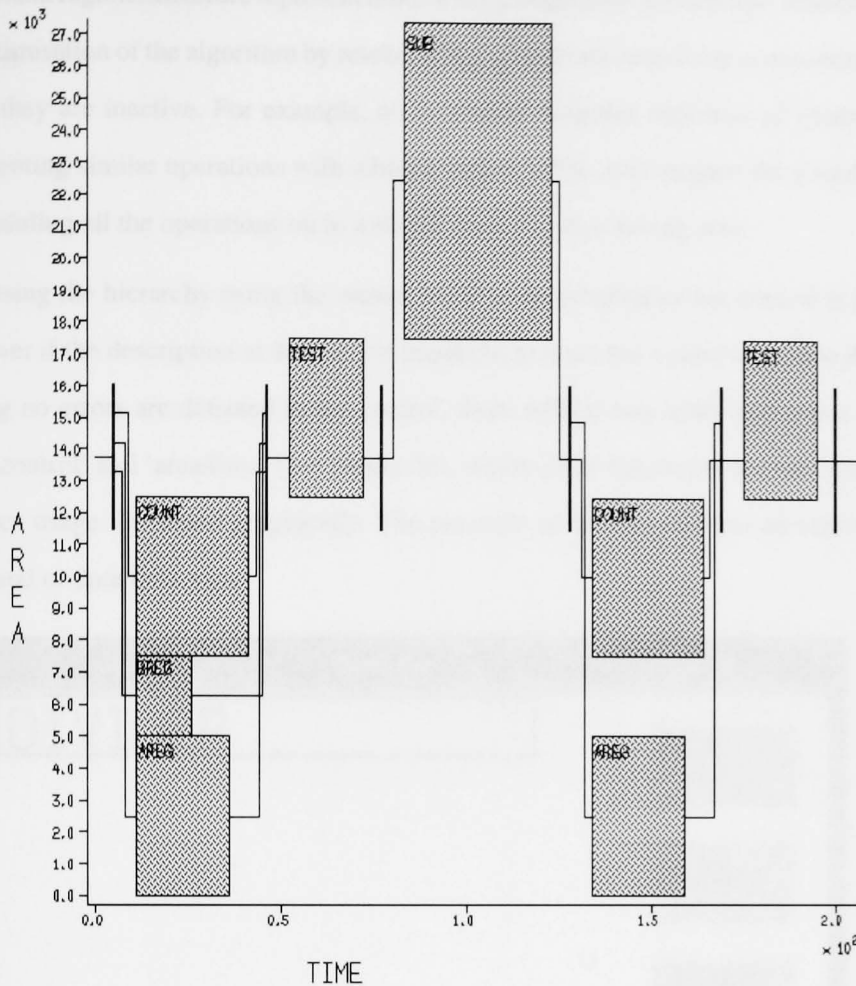


Fig. 25. Gantt resource allocation chart

In Figure 25 the algorithm of the divider described earlier is shown as it appears on the screen in this mode of the viewer. Time is plotted horizontally, and area vertically, with all the components being represented by rectangles scaled according to their computation time and area. If, as in this case the total time required for the control is dependent on the value of the data, the user will be asked to define a maximum time in the behavioural description so that resource allocation diagrams using that block can be displayed. Similarly, loops in the control must have the number of iterations defined by the user before display. In the diagram of Figure 25, each invocation of an operation performed in a particular component is shown shaded, and each individual component occupies a specific section of the area

axis. Blank regions therefore represent times when components are inactive, and may allow for optimisation of the algorithm by rescheduling operations on to those components at the times they are inactive. For example, an ascending diagonal sequence of shaded areas representing similar operations with a blank region below may suggest the possibility of rescheduling all the operations on to a single resource, thus saving area.

Traversing the hierarchy using the viewer is the same whether or not control is present. However if the description at the current hierarchical level has a control section then, assuming no errors are detected in the control, there will be two additional menu options 'view control' and 'area/time' (see Figure 26), which allow the user to see the control and resource usage illustrated graphically. The presence of control also has an effect on the structural or data flow view.

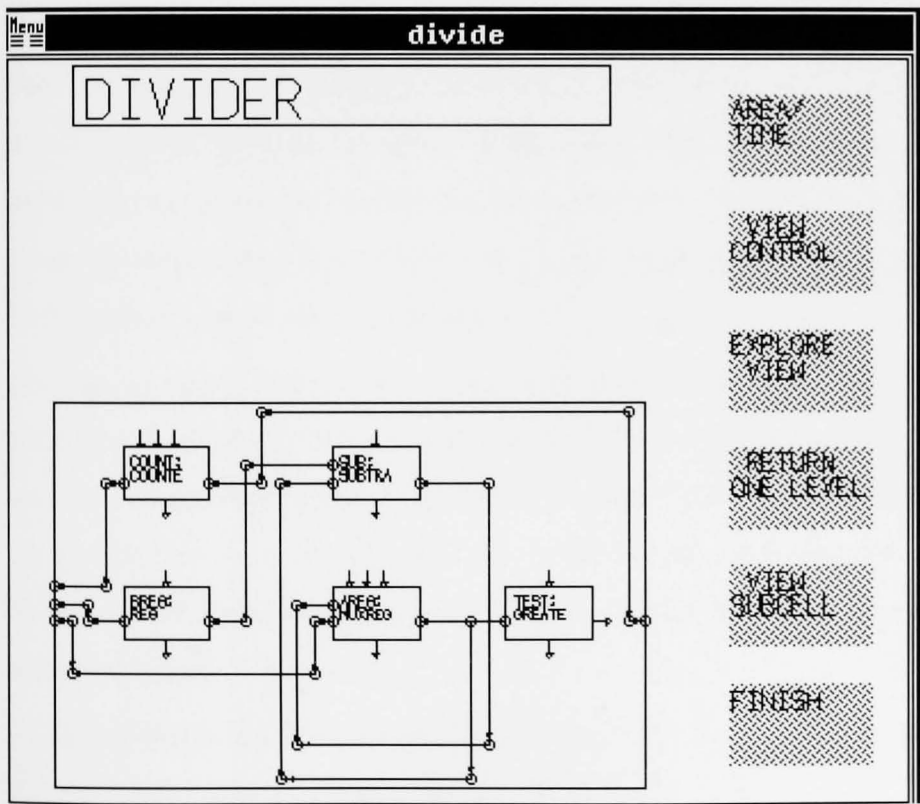


Fig. 26. Main screen – control section present

10. BOYER-MOORE INTERFACE

10.1. Introduction – formal methods

The Boyer–Moore interface tool was written to address one of the major aims of the STRICT language and system – correctness. As we have stated before, the main reason for choosing the functional programming paradigm for STRICT was the belief that such a notation would be amenable to formal verification of correctness.

Correctness is one of the major problems in VLSI design. VLSI designs must be correct before submission for fabrication, because fabrication is very expensive and takes a long time. Faults cannot be corrected on the chip, unless very special provisions are made, but this is unusual. Traditionally, simulation has been used to gain confidence in the correctness of a design, but simulation can no longer exercise design models completely due to the very large number of possible states[54]. As a result, formal methods have become a very active research area. The tutorial paper by McFarland[50] presents a good overview.

Formal methods in VLSI design are based on logics with a sound mathematical basis[8, 11], for example first order and higher order logics[33, 36]. Frequently, use is made of a theorem prover [7] for the manipulation and verification of formulae in the chosen language. The aim is to prove in a symbolic manner that a design meets its specification, thereby doing away with the need for simulation.

There are a number of theorem provers available, including Boyer–Moore[9], HOL[33], Nuprl[5] and EHDM[60]. Of these, Boyer–Moore and HOL both have a track record in the area of hardware design; Hunt[40] has used it to describe and verify microprocessor designs, while Cohn has used HOL to verify some properties of the Viper chip[20]. The work by Hunt and Cohn is known as *post hoc verification* – the work was performed on an already completed design.

The Boyer–Moore and HOL provers were available in the public domain, so we investigated these for possible use with STRICT.

10.2. Boyer–Moore versus HOL

The relative advantages and disadvantages of the Boyer–Moore and HOL theorem provers, as perceived in the literature (e.g. [32, 33] about HOL and [9, 57, 58, 66, 67] about Boyer–

Moore), software demonstrations, personal use and from comments made by several experienced users of these systems, are as follows.

The formalisms underlying both Boyer–Moore (first order logic, using a variant of LISP) and HOL (higher order logic, using ML) both have formal semantics, and both systems have been used for proving correctness of hardware designs. Both systems are relatively free of bugs. Both systems are widely used in the research community.

The Boyer–Moore prover attempts to do a proof without interactive user input, using built-in heuristics and libraries of theorems. Its output is considerable, and it attempts to give as much explanation as possible about the proof. Documentation about the system is excellent, provided in a book [9] which explains both the logic and the system in some detail. The input language is easy to learn, and a non-expert user can quickly get to grips with it. The learning curve is therefore not very steep.

HOL, by contrast, is an interactive system that requires the user to guide the proof through its various stages. No attempt is made to perform the proof automatically. Its output can only be described as terse, and there is no detailed documentation about the system. This means that HOL is much more difficult to come to grips with for a non-expert, and the learning curve is quite steep. All of this was deemed to be a major disadvantage in the STRICT environment.

It was therefore decided to investigate the Boyer–Moore prover more closely.

10.3. The Boyer–Moore prover

The Boyer–Moore theorem prover was initially developed at the University of Austin as a tool for research in Artificial Intelligence applications. It has been successfully applied in such diverse areas as list processing, number theory, protocols, real time control, and concurrency. The largest proof performed to date is that of Goedel's Incompleteness Theorem (more than 20,000 lines of proof code). The prover was used by Hunt [40] to verify the correctness of a microprocessor design, thereby confirming that it could be used in the VLSI design area. It is written in Common Lisp, and is distributed with a specially modified

version of the Kyoto Common Lisp interpreter. The software runs on a wide range of machines.

The Boyer–Moore logic is a quantifier free, first order logic with equality and function symbols. The input language to the prover is a variant of Pure Lisp. This input language is used to present the prover with a sequence of so called events. The most important events are: definitions of (often recursive) functions to be used in proofs, and theorems. The keyword **defn** starts the description of a function; all functions must conform to certain rules with respect to termination, which are checked by the prover before accepting the function definition. If a function calls other functions, these must have been previously defined. The keyword **prove-lemma** initiates the description of a theorem. A theorem will be acceptable if it can be proven by the prover. The prover will attempt to do this using its built-in logic inference rules, definitions, and previously proved theorems. It is in general necessary to build a database of carefully chosen and carefully ordered theorems to perform complex proofs.

As an example, we can define a function 'double' which doubles an integer by adding the integer to itself, and we then prove that this is the same as multiplying by 2:

```
(defn double (x)
  (plus x x))

(prove-lemma double-is-times-2 (rewrite)
  (implies (numberp x)
    (equal (double x) (times 2 x))
  )
)
```

The theorem includes the hypothesis that 'x' must be a number, and requires it to be stored as a rewrite rule. After it has been proved, the prover will subsequently replace occurrences of '(double x)' by '(times 2 x)'.

The prover has a number of powerful built-in heuristics, one of which is the capability of performing mathematical induction. This means it can cope most efficiently with recursively defined functions. Since recursion is used in STRICT specifications to achieve iteration, the fact that Boyer–Moore seems to cope well with recursion appears to be a clear vindication of the decision to introduce this feature in the STRICT language. The prover

is also distributed with a number of databases of useful functions and theorems which it frequently uses in proofs. These can be loaded at the request of the user.

The prover has a facility whereby the user can define his own abstract data types. This is usually referred to as the *Shell principle*. For the purpose of hardware verification, Hunt [40] has added a data type for use with bit vectors, and a whole set of associated theorems. These can be loaded automatically when the prover is started.

If the prover fails to complete a proof, this does not necessarily mean that there is something wrong with the theorem. It may be that the proof is rather complex, and that the prover cannot 'see' the right path through the proof. In this case it will be necessary to 'educate' the prover, by first proving smaller theorems which prove parts of the big proof, so that it will subsequently use these to complete the big proof. In many cases these smaller theorems will be useful in themselves, and can be added to the user's database if he wishes. The prover will therefore frequently need help from the user, which means that in a sense it is not fully automatic.

The 'prove-lemma' construct can have as an optional parameter a list of hints, which can tell the prover to use certain theorems with certain parameters, ignore other theorems, or perform inductions in a specified manner. Some of these features are shown in the example which now follows.

A special sub-system allows the prover to be used interactively, in a similar way to HOL. The important advantages of Boyer-Moore therefore appear to be automatic operation, excellent documentation, short familiarisation time, and the fact that it comes with large hardware database.

We now take a closer look at a proof example.

10.4. A proof example in Boyer-Moore

We want to prove that for all positive n the integer quotient of $n \cdot x$ and $n \cdot y$ is equal to the integer quotient of x and y , i.e. we will try to prove that

```
(equal (quotient (times n x)
                 (times n y))
```

```
(quotient x
          y))
```

Such a theorem occurs frequently when trying to prove properties of bit vectors, with 'n' having the value 2. We will call this theorem 'quotient-times-times'. Trying to prove it without the aid of additional theorems fails – the prover tries to solve the problem by using induction, which is not the right approach.

A substantial portion of Hunt's database was loaded first. The recursive definition of 'quotient' is [9]:

```
(defn quotient (i j)
  (if (equal j 0) 0
      (if (lessp i j) 0
          (add1 (quotient (difference (i j) j))))))
```

That is, the quotient is performed by repeated subtraction, while counting the number of subtractions performed.

The proof can be split up in three cases: $x < y$, $x == y$ and $x > y$. The first case is described by the following lemma:

```
(prove-lemma q1 (rewrite)
  (implies (and (greaterp y x)
                (greaterp n 0)
                )
    (equal (quotient (times n x) (times n y))
           (quotient x y)
    )
  )
)
```

The pre-defined arithmetic rules are not sophisticated enough to cope with this, so we first prove another lemma to tell it that if $x < y$ then $n*x < n*y$:

```
(prove-lemma q1-first (rewrite)
  (implies (and (greaterp y x)
                (greaterp n 0)
                )
    (greaterp (times n y) (times n x)
    )
  )
)
```

Now lemma 'q1' is proved, using 'q1-first' and the pre-defined rules. The case $x == y$ is easily proved from pre-defined rules:

```

(prove-lemma q2 (rewrite)
  (implies (and (equal y x)
                (greaterp n 0)
                )
    (equal (quotient (times n x) (times n y))
           (quotient x y)
    )
  )
)

```

The third case, $x > y$, is the most general one and fails if tried on its own. It can be proven if one realises that if $x > y$, then $x = a*y + b$, where $a > 0$, and $b < y$. Thus we first try the following lemma:

```

(prove-lemma q3-first (rewrite)
  (implies (and (greaterp a 0)
                (greaterp y b)
                (greaterp n 0)
                (equal (plus (times a y) b) x)
                )
    (equal (quotient (times n x) (times n y))
           (quotient x y)
    )
  )
)

```

The proof of this lemma is achieved by using various database rules. In order to prove the lemma for the case $x > y$ we need to tell the prover to use this lemma, while substituting '(quotient x y)' for 'a', and '(remainder x y)' for 'b'. This will allow the prover to check that the hypothesis '(greaterp x y)' holds. Note the use of a hint in this lemma:

```

(prove-lemma q3 (rewrite)
  (implies (and (greaterp x y)
                (greaterp n 0)
                )
    (equal (quotient (times n x) (times n y))
           (quotient x y)
    )
  )
  ((use (q3-first (a (quotient x y))
                  (b (remainder x y))
                )
  )
)
)

```

The main lemma still cannot be proven – we have to tell the prover that the previous three cases together constitute any possible combination of x and y :

```

(prove-lemma q-bridge (rewrite)
  (implies (and (greaterp n 0)
                (or (greaterp y x)
                    (equal y x)
                    (greaterp x y))
            )
    (equal (quotient (times n x) (times n y))
           (quotient x y)
    )
  )
  ((use (q1))
   (use (q2))
   (use (q3))
  )
)

```

We are now ready to do the final proof:

```

(prove-lemma quotient-times-times (rewrite)
  (implies (and (greaterp n 0)
                (numberp x)
                (numberp y)
            )
    (equal (quotient (times n x) (times n y))
           (quotient x y)
    )
  )
  ((use (q-bridge)))
)

```

After the proof has been completed, all but the last lemma should be disabled, since they are only a special case of 'quotient-times-times'. In other proof efforts, some might be useful, and should therefore be kept. [9] contains several examples. Unfortunately, it is not possible to provide a general rule to determine whether a lemma should be kept or not.

This example shows the process of educating the prover. It should be obvious that it is very difficult to automate this.

10.5. Mathematical proof of hardware specifications

The question is now: how do we prove the correctness of a hardware module using the Boyer-Moore prover? The problem can be stated as follows:

given a block, its specification, its immediate subblocks and its structure, and the specification of the subblocks, *verify that* the sub-block specifications, composed as specified by the structure of the top level block, form the top level specification.

By performing this proof at all levels of the hierarchy, the top level specification of the entire design has been proved. Since we have made sure that STRICT allows hierarchical designs which contain behavioural specifications at all levels, such a proof is possible. The major problem is (as demonstrated in the proof example of the previous section) that the resulting proof may be too complex for the theorem prover at the higher levels of the design, and that manual intervention of the designer is required. This is not an ideal situation, since designers are likely to lack the highly specialised skills required.

10.6. Extraction

The interface uses the procedural interface to the parse tree. The resulting data structure is then used to extract all relevant information.

10.7. Output

We demonstrate a correctness proof by means of an example. Shown below is a typical module, a half adder which is composed of an AND gate and an XOR gate:

```
block halfadder
  having (x, y: wire):
    (out: wire[2])

    intended behaviour
      whenever
        change(x) or change(y):
          within (22)
            set out = plus(x,y);
  use structure
  { instance
    a1: andgate2
    x1: xorgate
  using
    a1(x,y)
    x1(x,y)
```

```

        make
            out ::= join(wire[2]|a1.out,x1.out)
        }
end

block andgate2
having (a,b: wire) :(out: wire)
    intended behaviour
        whenever
            change(a) or change(b):
                within (9)
                    set
                        out = v_and(a,b);
end

block xorgate
having (a,b: wire) :(out: wire)
    intended behaviour
        whenever
            change(a) or change(b):
                within (9)
                    set
                        out = v_xor(a,b);
end

```

The Boyer–Moore logic requires all input to be defined before use, so the logic is generated in a bottom–up fashion. The following sections are generated in turn:

- all function definitions from the subblocks;
- behavioural specifications of the subblocks. This requires application of the behavioural statements to the appropriate input busses, followed by conversion into an integer using the Boyer–Moore function 'bv-to-nat' (which converts bit vectors to natural numbers);
- all function definitions from the top level;

- the high level specification. This does not require the use of 'bv-to-nat';
- the top level structure, as defined in the 'using' and 'make' sections from the STRICT description;
- the final prove-lemma that asks the prover to carry out the actual proof.

Note that some of these sections may be missing, since not all are mandatory in the STRICT description.

10.8. Results

The half adder example generates the following Boyer-Moore input file.

```
(proveall "bm" '(
;;;;; defs from low level WHERE
;;;;; low level specs
(defn xorgate-spec (a b)
  (bv-to-nat (v-xor a b))
)
(defn andgate2-spec (a b)
  (bv-to-nat (v-and a b))
)
;;;;; defs from high level WHERE
;;;;; high level spec
(defn halfadder-spec (x y)
  (plus x y)
)
;;;;; defs corresponding to USING
(defn a1 (x y)
  (andgate2-spec x y))
(defn x1 (x y)
  (xorgate-spec x y))
```

```

;;;;;; high level structure from MAKE

(defn halfadder-circuit (x y)
  (bv-append (nat-to-bv(x1 x y) 1)
             (nat-to-bv(a1 x y) 1)
  ))

;;;;;; finally, the proof

(prove-lemma halfadder-circuit-ok (rewrite)
  (implies (and (bitvp x)
                (bitvp y)
                (equal (size x) 1)
                (equal (size y) 1)
              )
            (equal (bv-to-nat (halfadder-circuit x y))
                    (halfadder-spec (bv-to-nat x)
                                     (bv-to-nat y))))))

;;;;;; end of proof

))

```

The following should be noted about this.

- Some of the sections are empty, because the corresponding STRICT sections are empty.
- Because of the functional notation used in STRICT, translation of functions to Boyer–Moore format is quite straightforward – which again vindicates our belief in the functional notation employed in STRICT.
- Similarly, because the 'using' and 'make' sections are written in a functional form, their translation to Boyer–Moore format is also very simple. It is not necessary to decompose the net structure, since the built-in heuristics will do all the work. For example, if the function 'head' is used anywhere in the using section, the first step taken by the prover is to apply this function, as part of the simplification heuristic.

The (partially reproduced) result of feeding the above statements into the prover is shown in appendix A.

Unfortunately, only when very simple examples such as the half adder are used can the prover complete the required proof on its own. When more complex examples are attempted, the user must generate hints to lead the prover through the proof, which requires a great deal of effort. Hunt's thesis gives an excellent impression of the work required.

Because there is no obvious way in which such hints can be generated automatically, as also reported in [56], it was concluded that post-hoc verification is perhaps not the most convenient verification method. The next chapter shows how correct transformational synthesis can be achieved using the Boyer-Moore prover.

11. THE TRANSFORMER

11.1. Introduction

The transformer tool represents the second major aim of the STRICT language and system: to have the ability to generate a formally correct implementation of a high level behavioural description, a process known as High Level Synthesis [49].

The high level synthesis community is split into two camps: one which wants to fully automate the design process, and one which wants to make use of the designer's experience by providing interactive input during the design process. In fully automatic systems, much of the current research is focussed on the provision of algorithms to ensure that reasonably efficient hardware is generated from a large number of possible implementations. Typical research efforts are reported in [2, 12, 17, 38, 43, 44, 48, 55, 61]. The interactive approach encourages designers to investigate different designs through experimenting with different architectures generated from the same specification by, for instance, serialising operations upon the same device or by allocating many operations to devices operating in parallel. The most important effort in this area is the SAGE tool [22], but SAGE does not support verification. Other important tools are LAMBDA[27] and VERITAS[37], which use polymorphic predicate calculus and type theory, respectively, as the underlying formalisms. We firmly believe in the interactive approach, but we would like to use the Boyer–Moore logic as the underlying formalism and the Boyer–Moore prover itself as the verification tool.

A specification of an algorithm in a high level language is likely to be unsuitable for direct translation into silicon. It is necessary to perform transformations[14] which preserve the behaviour, but generate a much more compact silicon implementation [49]. The problem is then to ensure that the applied transformations preserve the behaviour [26, 14], and in general to verify the correctness of the ultimate design produced by the system against the initial specification. Research in this area is reported in [66, 25, 27]. As mentioned in the previous paragraph, our aim was to use the Boyer–Moore theorem prover to achieve this goal.

11.2. Author's contribution

The author was deeply involved with the development of the ideas behind this tool, and for the parse tree interface that it uses. Programming of the tool was done by a research student. The ideas are again presented here in the form of screen dumps that show the output of the tool, when applied to the error corrector example of chapter 3.

11.3. Transformational synthesis

An excellent introduction to high level synthesis systems can be found in the overview paper by McFarland [49]. All high level synthesis systems initially operate upon a user specified behavioural description in an appropriate high level language. This description is parsed into an internal format. This is followed by the scheduling and allocation phase, during which the basic functional units of the design are determined and the basic hardware units are assigned to these functional units, together with memory elements and communication paths. The resulting design is then fed into conventional floorplanning and routing tools to produce the final chip layout. State of the art examples of high level synthesis systems are Cathedral [21] and the Yorktown Silicon Compiler [10]. No work using the Boyer–Moore prover has been reported. This chapter describes a transformational synthesis tool that integrates the Boyer–Moore prover.

11.4. Basic ideas

The basic idea behind the transformer tool is as follows. A Boyer–Moore rewrite rule specifies an equality. In other words, the left hand side of the equality can be replaced by the right hand side, and vice versa. We can therefore regard such a rewrite rule as a *correctness preserving transformation* when viewed from the hardware correctness point of view.

The idea is therefore to develop an interactive design tool that allows the designer only use of Boyer–Moore rewrite rules for introducing changes, thus guaranteeing the correctness of the design procedure.

11.5. Extraction

In the previous chapter we showed that the translation of STRICT behavioural specifications into the Boyer–Moore format is relatively straightforward. The transformational tool interfaces to the STRICT parse tree by using the appropriate procedural interface.

11.6. Design procedure

The following design procedure is followed.

- The transformer extracts a specification from the STRICT description through the procedural interface, and converts this into a functional tree, which is displayed on the screen.
- The transformer makes use of a number of libraries: a (fixed) library for basic Boyer–Moore rules, a similar user defined library (which the user can modify), and a library of basic cells with their functional specifications.
- The designer clicks on a node on the screen. The transformer will locate appropriate rewrite rules in its libraries, and show the results to the designer.
- The designer then chooses one of the rewrite rules. The transformer applies the rule, adjusts the functional tree, and displays the resulting tree.
- When the designer is satisfied with the design, he asks the tool to allocate hardware. Appropriate modules are located in the cell library, and a complete STRICT description is generated, completing the design cycle.

The result is that a complete structure can be interactively generated from the behavioural description, whilst guaranteeing that the final result still implements the original specification, because correctness preserving transformations were used during the entire design process.

The design process is clearly dependent upon the fact that, in STRICT, behavioural descriptions are mandatory, whilst structural descriptions are optional.

11.7. Overview of operation

The behavioural specification for each block is captured in a set of functional expressions, along with temporal information. The tool transforms this specification into a functional tree. This tree is drawn on a screen by a graphical subsystem with which the designer can subsequently interact. In order to ensure that all interactions preserve the correctness of the design, only changes that correspond to a set of formal transformations are allowed, and these must first have been verified by the Boyer–Moore theorem prover .

These transformations are kept in two libraries, one generated by the user of the tool (for use with the current user defined specification), and the other one a standard library of re-write rules which can be applied to the set of operators built into the STRICT language. Both libraries come in the form of a text file, and their contents must be acceptable to the Boyer–Moore prover (it is the user’s responsibility to ensure that they are). Both libraries are therefore generated separately, before the synthesis tool can be used. The standard library contains rules relating to the following STRICT operators:

	TIMES		GREATER		NOT
	ADD		CONDITION		AND
	MINUS		EQUALITY		OR
	DIVIDE		LESS		FUNCTION

Fig. 27. Library operators

The library is arranged in sections where each section corresponds to one of the nodes shown in Fig. 27. If, for example, the designer clicks on the + node on the screen, the rewrite rules relating to the add node are made available. A typical example of such a rule might be

```
(equal (plus a (plus b c))
      (plus a b c))
```


which simplifies the functional tree by removing one plus operation.

A behavioural specification in a high level language will usually not correspond to the most efficient hardware implementation. It will generally be necessary to modify its functional tree in order to improve the efficiency. Since only those changes can be made that have first been verified by the prover, all modifications are by definition correct. The theorem prover therefore ensures that all changes made are carried out within a formal framework. Once the designer has completed his work on the functional tree, hardware can be allocated, by fetching the appropriate modules corresponding to the various parts of the design from a hardware library.

When the transformations and hardware allocation have been completed, a STRICT description of the complete design is generated, and the final layout can then be generated using standard floorplanning and routing tools.

The interactive interface of the synthesis tool is shown in fig. 28. The functional tree for one level in the design hierarchy is displayed in the centre of the screen. Below it is its lisp description. Interaction with the functional tree takes place by clicking on the icons which are situated around the edges of the screen. To access the sub-trees of the functional tree, a small list of icons on the right hand side of the screen is available. It begins with 'eva' and ends in 'ret' (which stands for 'return', and which enables the user to move back up the design hierarchy). To carry out modifications to the tree the 'thsrch' icon at the bottom left of the screen is selected, followed by a node on the functional tree. The system responds by searching the rewrite rule library section associated with the chosen node and selects a list of applicable rewrite rules which are presented to the user as options. If the user wishes to carry out the modification, he selects it directly and applies the change. The 'store' icon which is above the 'thsrch' icon can be used to store a particular rule which the designer may wish to apply more than once. In this case the rule is passed to a small buffer where it can be selected and applied without searching the library for it.

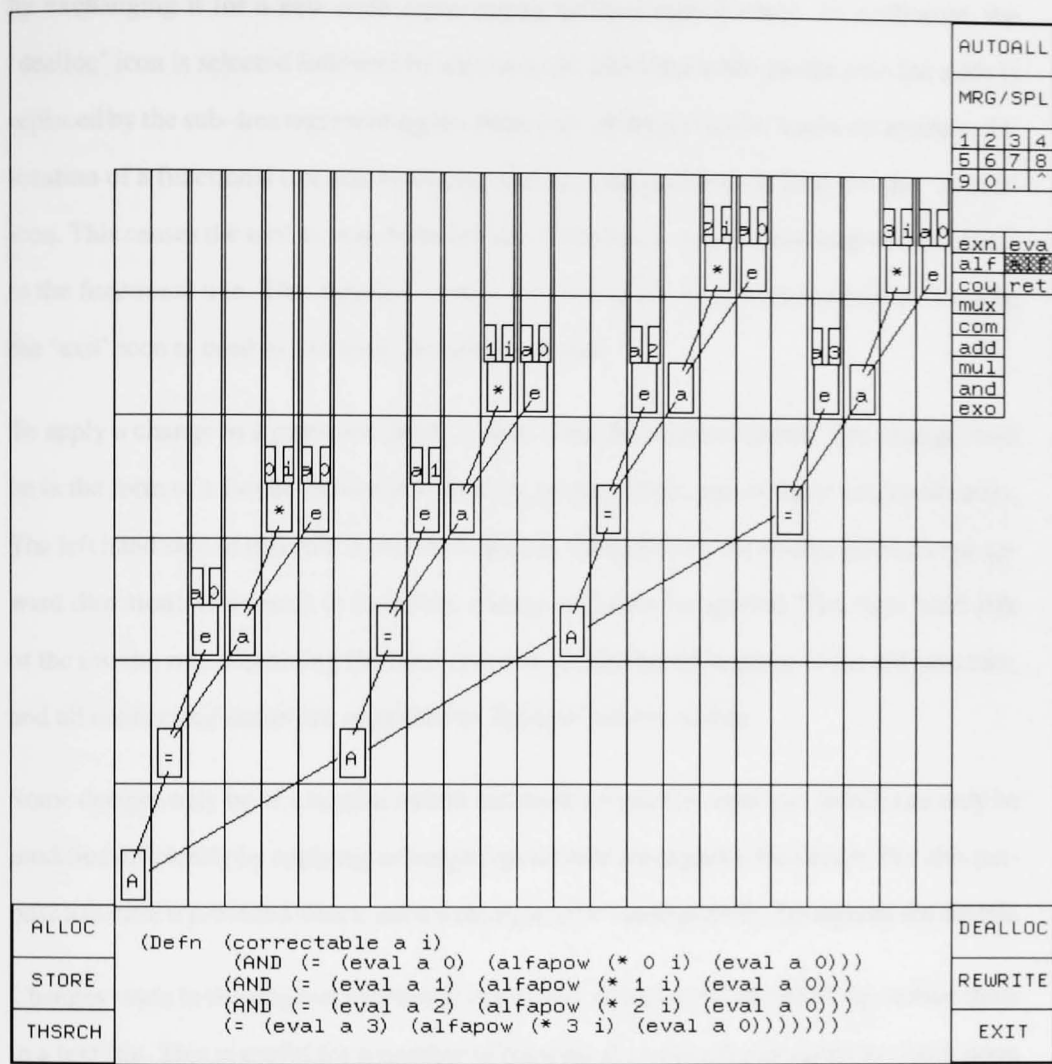


Fig. 28. Main screen

The 'rewrite' icon at the bottom right hand corner of the screen is used to rewrite a sub-tree within a functional tree. The 'alloc' and 'dealloc' icons are used to allocate hardware to the functional tree. Allocation for a particular node on the functional tree is done by clicking on the 'alloc' icon followed by the selection of a node on the functional tree. The system responds by searching a library of hardware modules, each of which has an associated behavioural description. A list of modules with area and time information is provided. These are guaranteed to implement the behaviour at the chosen node. The user then selects a mod-

ule, and the functional tree is modified at the node where the implementation is carried out, by exchanging it for a new node representing the hardware module. To deallocate, the 'dealloc' icon is selected followed by a previously allocated node. In this case the node is replaced by the sub-tree representing the behaviour of the particular hardware module. Allocation of a functional tree can be carried out automatically by clicking on the 'autoall' icon. This causes the tool to search the library of hardware modules and map them directly to the functional tree. The 'mrg/spl' icon is used for space-time transformations. Finally, the 'exit' icon is used to exit from the synthesis tool.

To apply a change to a particular point, a node from the tree is selected. The change must be in the form of a Boyer-Moore rewrite rule, selected from one of the available libraries. The left hand side of this rule is matched against the tree from the chosen node (in the upward direction). If a match is found the change will then be applied. The right hand side of the rewrite rule containing the new structure is substituted in place of the old structure, and all connecting nodes are appended to the new section of tree.

Some designs may be of a regular nature and have a repetitive structure which can only be modified efficiently by applying a change repetitively throughout the design. For this purpose a feature is provided which allows changes to be made globally throughout the design.

Changes made to the original functional tree by the designer are recorded, by storing them in a text file. This is useful for a number of reasons. It enables the designer to check upon his own modifications, once the design is completed. Also, whilst changes are being made to the original specification by applying rewrite rules, the theorem prover may generate new rules which the designer may wish to keep. For example, a larger rewrite rule may result from a series of smaller changes, or the designer may be able to derive a new rewrite rule. If such a rewrite rule is stored, it could easily be added to the user library at a later point.

Allocation of hardware is split into two stages, manual and automatic. The manual stage concerns the binding of operational units to the operators on the functional tree. For this purpose a library of hardware components is available, so that the designer can choose

from a possibly large set. Area and time information is shown in graphical form at the top of the screen as he chooses his components and allocates them.

Synthesis example

The screen of Fig. 28 shows an example function which was taken from a error decoder module described by Kalker [42] (the whole example is shown in section 3.9.8). The function is called 'correctable' and would be defined in STRICT as follows:

```
correctable(a: byte[32], i: integer):BOOLEAN :=  
    (eval(a, 0) == alfapow(0*i, eval(a, 0))) AND  
    (eval(a, 1) == alfapow(1*i, eval(a, 0))) AND  
    (eval(a, 2) == alfapow(2*i, eval(a, 0))) AND  
    (eval(a, 3) == alfapow(3*i, eval(a, 0)))
```

The bottom of the screen shows the S-expression equivalent of this function. We demonstrate the operation of the tool by showing how one can make this function more efficient by making formal transformations.

Any modifications that might be applied to the tree rely upon the use of rewrite rules from the library, such as:

```
1 / (equal ((times i 0) 0))  
2 / (equal (equal a a) t))  
3 / (equal ((and t a) a))  
4 / (equal (and a (and b c))  
        (and (and a b) c)))
```

In the rest of this section we will frequently refer back to these rewrite rules.

We turn our attention to the leftmost sub-tree on the screen of Fig. 28. After clicking on the multiplication node, the screen of Fig. 29 appears.

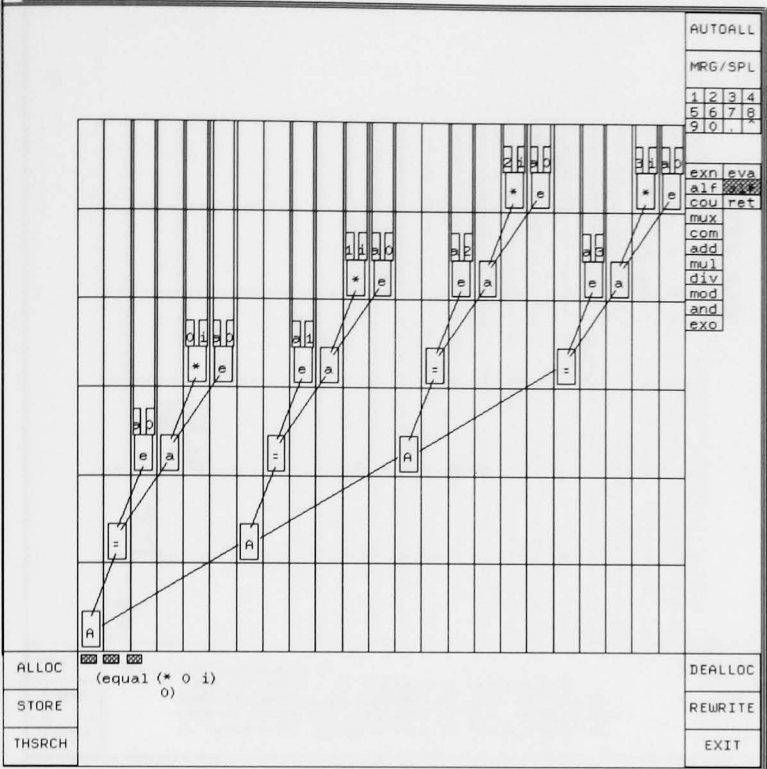


Fig. 29. After click on leftmost multiplication

Near the bottom of the screen 3 boxes have appeared, indicating that three rewrite rules from the library are applicable at this point. The first of these rules is printed below the boxes, and, by clicking on each of the boxes in turn, the designer can cycle through them. Application of rule 1 (which says that $0*i$ equals 0) will result in deletion of the multiplication sub-tree. This is shown in Fig. 30.

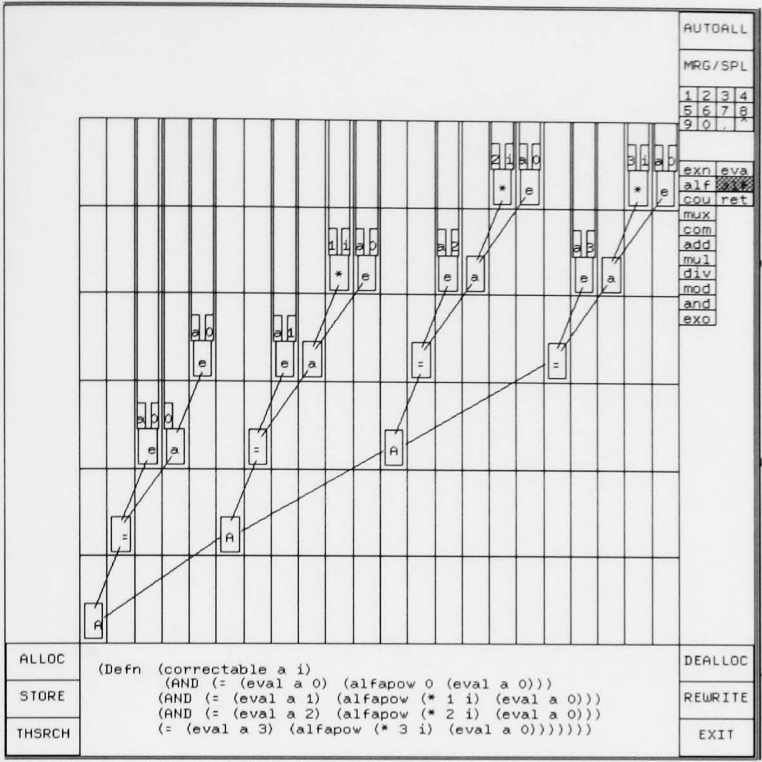


Fig. 30. After deletion of multiplication sub-tree

We then have a sub-tree representing $\text{alfapow}(0, \text{eval}(a, 0))$, which can be replaced by $\text{eval}(a, 0)$ by rewriting. The rewrite rule is shown at the bottom of Fig. 31, and the result of the application of the rule in Fig. 32.

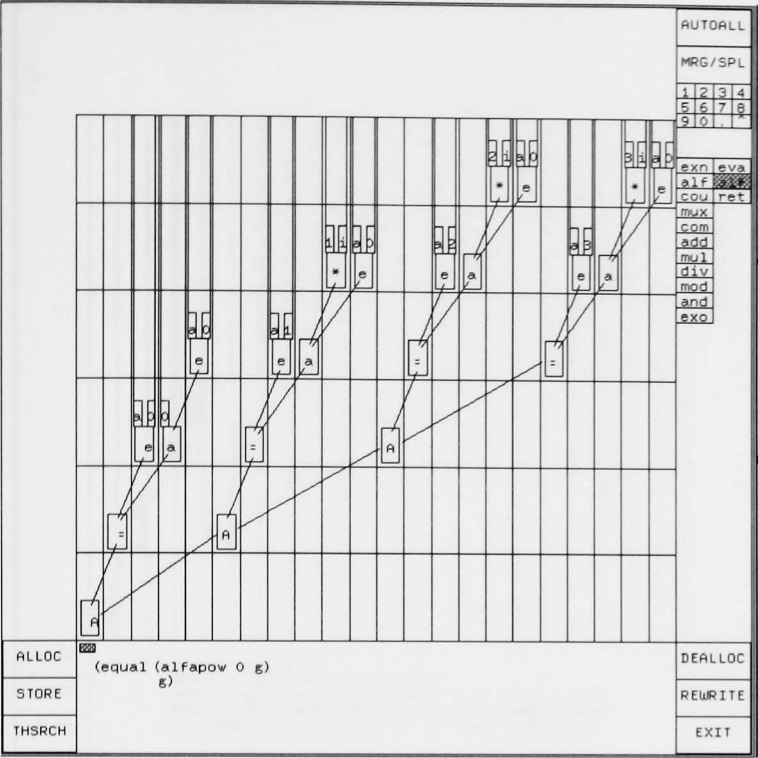


Fig. 31. Showing rewrite rule

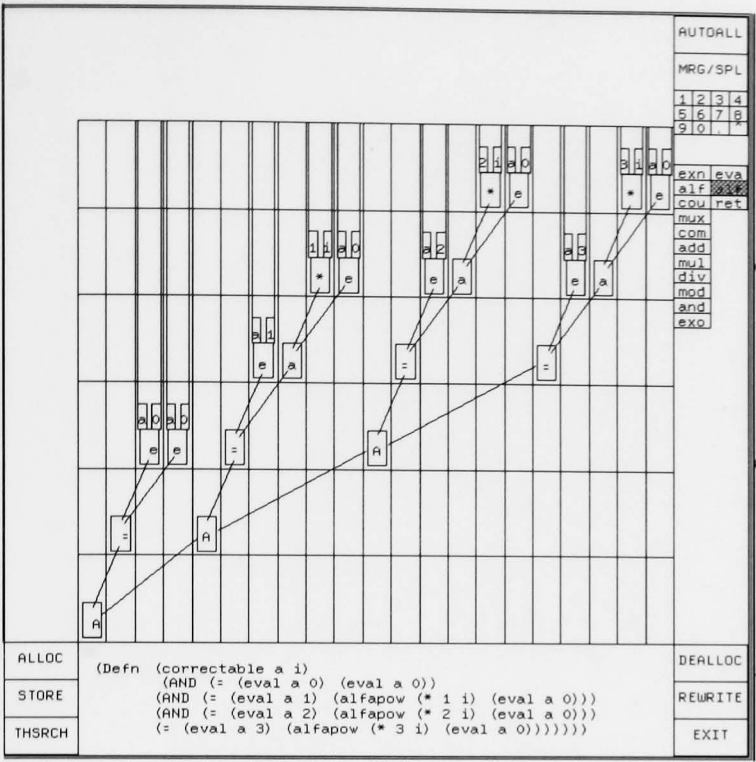


Fig. 32. Applying rewrite rule

The resulting sub-tree has two equal branches, so rule 2 can be applied (see Fig. 33) to the '=' node to give the result T. This is shown in Fig. 34.

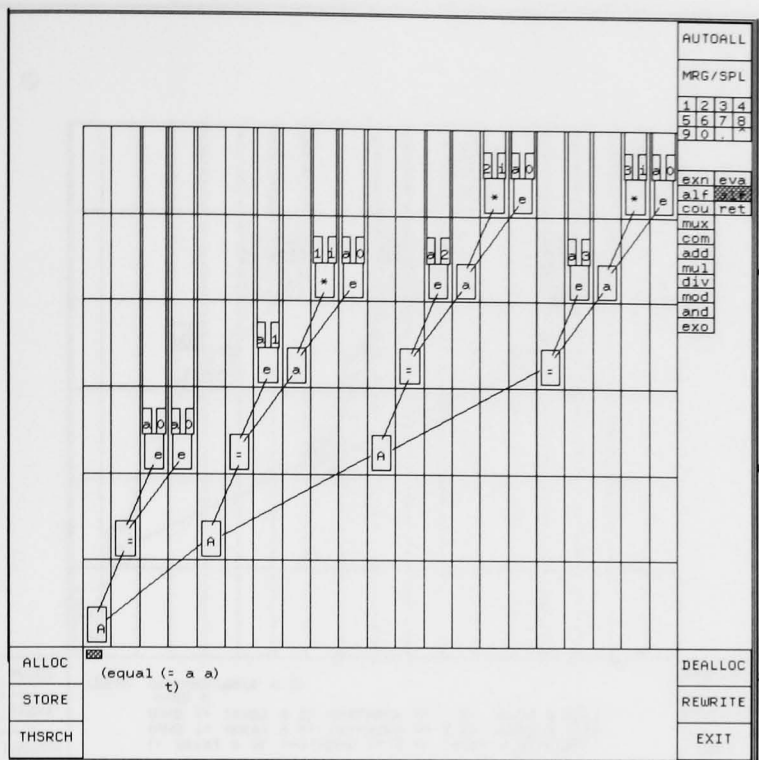


Fig. 33. Applying rule 2

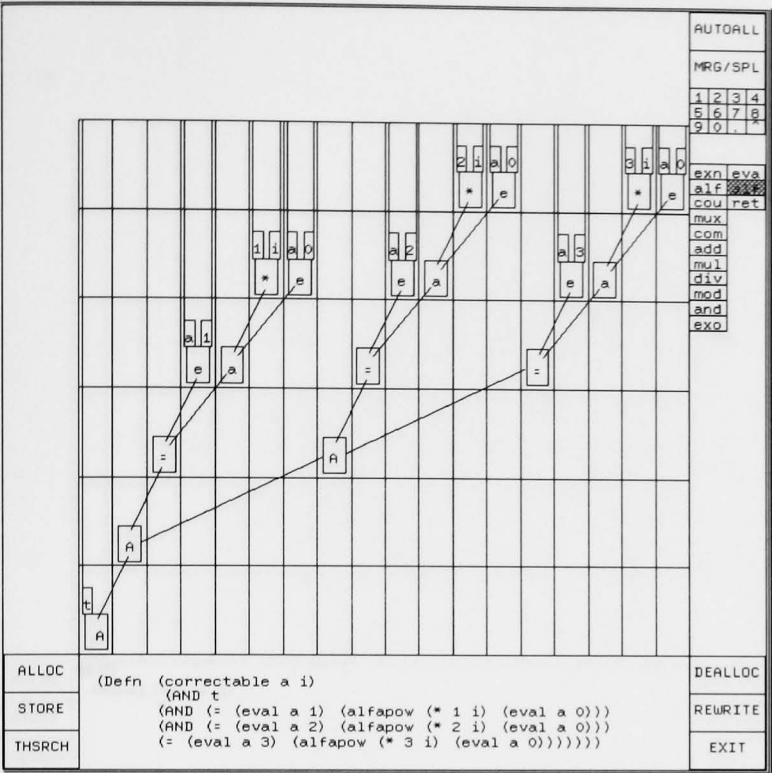


Fig. 34. Result of rule 2

Since ANDing with T is a no-op operation (see bottom of Fig. 35), rule 3 can be applied to the 'and' node, resulting in the removal of the leftmost sub-tree.

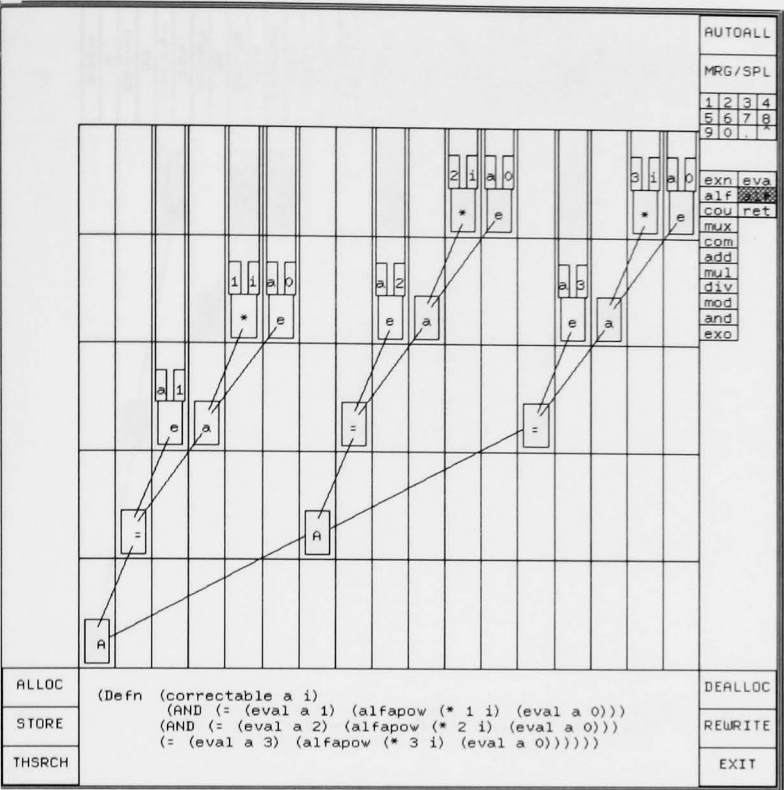


Fig. 36. Final result

At this point, it is possible for the designer to allocate actual hardware. This is shown in Fig. 37, where nodes that have been allocated are shown as shaded boxes. At the top of the screen, for each box an estimate of the area and the speed for each node is shown. These in turn may prompt the designer to select a particular node, for example, one with a very large area, in order to do further optimisation.

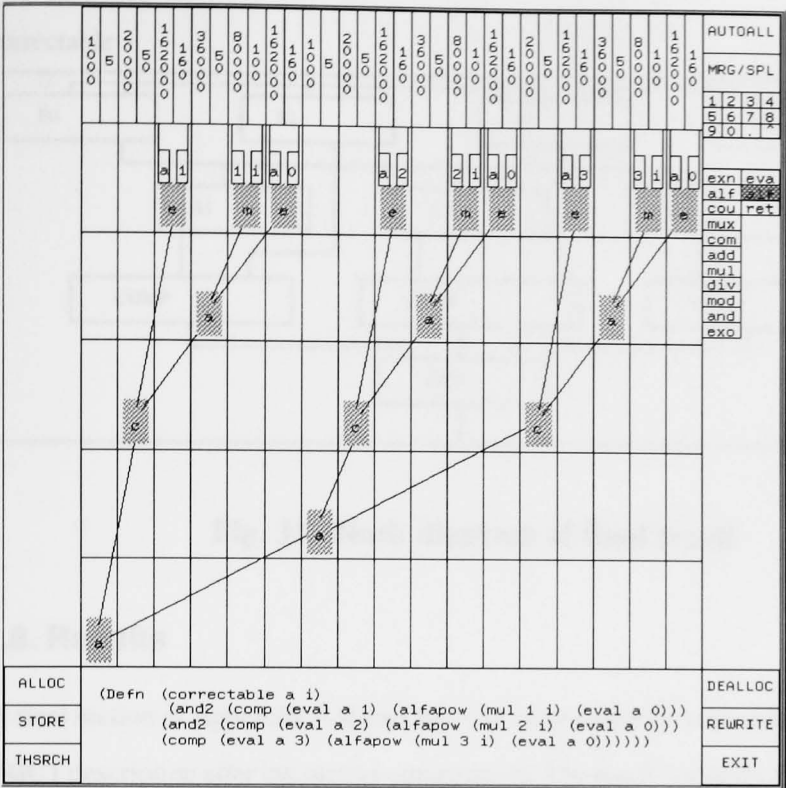


Fig. 37. Hardware allocation

A block diagram of the resulting implementation is shown in Fig. 38. We have abbreviated eval(a,0) to E0 etc., and alfapow(1*i, eval(a, 0)) to A1 etc. Upon exit, the tool will generate a complete structural STRICT description of the design, which can subsequently be used to generate its final layout.

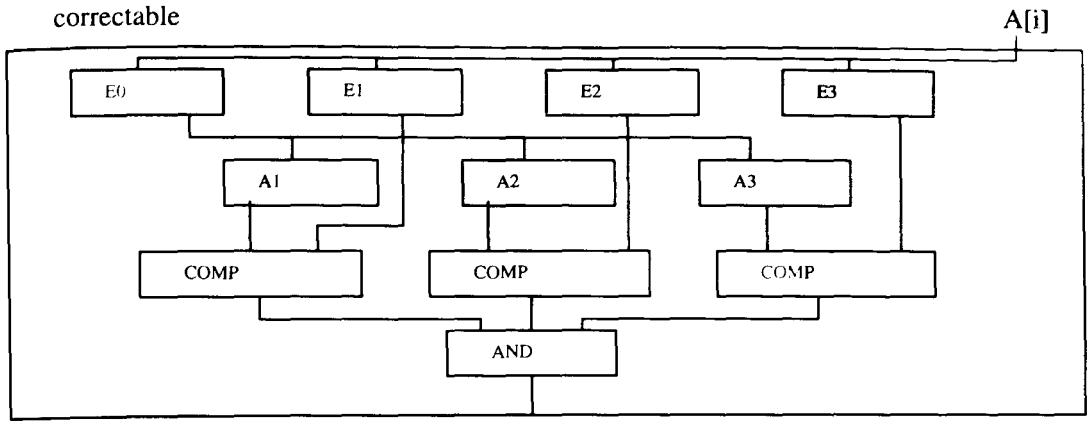


Fig. 38. Block diagram of final result

11.8. Results

The final section of Appendix A shows how the error corrector is generated as a structural STRICT description after interactive optimisation. The result is a fairly complex structure, which is assured to implement the stated behaviour through the use of formal transformations. The structure can be subjected to layout generation in the usual manner.

12. CONCLUSIONS

12.1. Introduction

Inevitably, the work presented in this thesis shows both advantages and disadvantages when compared with other work. We first argue the case in favour of the language and system, and then describe how the language could be improved.

12.2. Advantages

We specifically set out to investigate the use of a functional notation, using recursion equations with added timing information. Iteration was only allowed to be achieved through the use of recursion. The expectation was that this would allow formal verification and synthesis of descriptions in the language, features which were simply not widely available in HDLs at the time the language was developed. The functional notation was to allow the capture of behaviour, structure and control flow in a hierarchical manner. Behavioural specifications was to be mandatory, structural decompositions optional. We added to this a list of desirable features, presented in chapter 2: hierarchy, library facilities, generic features, data abstraction, strong typing, arithmetical operators, timing, synchronous and asynchronous design descriptions, and interfaces for module generators. The language, through the use of a number of procedural interfaces, was to fit into a sophisticated design environment, comprising the following tools: a syntax directed editor, a graphical viewer, a simulator, a layout system, a module generator, a theorem prover and an interactive transformation tool.

Most of the above aims have been achieved. We believe that we have also shown that the language is clearly superior in some respects to VHDL, Verilog and ELLA.

A description in the language is a collection of blocks, each with mandatory behaviour and optional structure, as determined by a set of appropriate keywords. The functional aspects of the language come in the form of functions that can be specified in a number of places:

- as part of the block declarations,
- in type declarations,
- at the end of the behaviour section.

These functions are subject to scoping rules; the last two kinds of functions are regarded as local, and the first kind global. These functions all require the use of recursion to achieve iteration. In addition, the syntactic forms of structural aspects of the language, such as block declarations, instance declarations, and the specification of netlists, have all been chosen to resemble function calls.

Other features of the language fit into this framework, through the use appropriate keywords. All information about a block is normally present within the same textual unit (unlike VHDL), the only small exception being declarations of types and functions, which may optionally be imported from other blocks or from libraries.

Let us examine the language in the light of the list of requirements specified in chapter 3.

The language obviously describes both behaviour and structure. We have taken the view that behaviour and structure should be completely separate (unlike ELLA), in sections denoted by the keywords **behaviour** and **structure**. Specification of behaviour is mandatory, whilst specification of structure is optional, reflecting the aim of allowing automatic translation of behaviour to structure using appropriate design tools. The STRICT timing model (see for example sections 3.9.2 and 3.9.9) is a simple 'cause and effect' model that is easy to understand. A large class of behaviours can be expressed effectively using such a model. By contrast, the VHDL model has different kinds of delays (e.g. inertial and transport) whose effects must be understood in terms of a stack of output drivers which may cause cancellation of previous events, possibly in the context of multiple concurrent processes. This takes a lot of effort to understand and to learn to use, but allows very accurate modelling of very complex circuits. The STRICT timing model is much more flexible than the ELLA timing model. However, it also has disadvantages, which are discussed below. The way finite state machines are described in the language (see the traffic light controller example, section 3.9.9) is rather similar to the way this is done in the other languages, so we cannot claim any major advantage here. The reasoning for choosing a particular syntax for the FSM model are again discussed below.

The language allows designs to be built hierarchically, by specifying a collection of blocks. Each block is started by the keyword **block**. The block specifies a number of named ports (input, output or inout) which allows the block to communicate with the outside world through connections to ports on other blocks (or, in the case of the top level of the design, to the output pins of the chip). By allowing blocks to instance other blocks, a hierarchical tree of blocks is created.

Each level in the hierarchy allows simulation, even if the level below is incomplete.

The language allows one of the main aims: formal verification and formal synthesis. In fact, this comes naturally with the language. The use of recursive behavioural functions (see for example the error corrector, section 3.9.8) has allowed us to do this efficiently. Through the Boyer–Moore interface, STRICT descriptions can be fed to a theorem prover, and, for simple examples, a fully automatic proof of correctness can be obtained. This is entirely due to the fact that the Boyer–Moore prover was specially written for use with functional languages such as LISP, and, more specifically, handles recursive functions quite well. The functional form of 'using' statements allows us to feed them in most cases unchanged to the prover, a major benefit of the chosen notation. The prover interface clearly vindicates claims that functional programs are amenable to formal verification techniques. The Boyer–Moore interface is a major positive point for the STRICT language, even though development of the interface was abandoned when it became clear that limitations of the prover heuristics would prevent it from handling complex proofs. This is currently a general problem with theorem provers, but the situation can be expected to improve with time. We have also shown that the transformer subsystem allows, in conjunction with the Boyer–Moore prover, the generation of a correct implementation of a functional behaviour, without the need for the designer to specify any form of structure. Boyer–Moore rewrite rules allow correct formal interactive changes to be applied. The transformer subsystem is another major positive point for the STRICT language.

We have shown examples (albeit rather simple ones) of how the language allows traditional techniques such as layout and simulation to be applied, with satisfactory results. Whether or not the results are competitive with those obtained from other design tools falls outside

the scope of the thesis; however, it can be expected that highly sophisticated design tools such as those from Berkeley will produce a more compact layout than the GAELIC system. This can only be expected, given the relative amounts of effort invested in these tools. The language constructs for control modelling allow high level graphical views, including Gantt charts. To the best of our knowledge these are a novel feature, and have proven to be very popular with users.

We strongly believe that STRICT is more concise than VHDL. Structural descriptions in STRICT are very concise and well integrated (see, for example, sections 3.9.2 and 3.9.3). Every block must include the header with information about its ports – there is no separate architecture declaration. The functional form of the 'using' statement is very concise. There is no need, when declaring components, to repeat port names, as in VHDL (which adds considerably to the verbosity of VHDL). The use of recursion in STRICT, particularly when describing parameterised cells, allows very concise descriptions of large and possibly complex blocks (for example, the n -input OR gate of section 3.9.4). In addition, the semantics of behaviour in STRICT are much easier to understand than those of VHDL. STRICT also has explicit constructs for modelling control. The syntax of this part of the language is quite concise. The other languages discussed in this thesis do not have special control features, although it would be possible, for example, to extract control information from VHDL descriptions if appropriate behavioural descriptions were present. STRICT is more readable than ELLA. This can be seen by comparing the recursive Sigma functions in both languages: the ELLA version from section 2.3 and the STRICT version from section 3.9.7. In contrast to ELLA, STRICT clearly separates the behaviour from the structure. This allows different graphical views of both to be presented, increasing the understanding of the design. The requirement that behaviour must always be present makes it easier for the designer to avoid the temptation of thinking straight away in terms of hardware implementation, thereby possibly avoiding incorrect design decisions early in the design process.

STRICT supports abstract data types, using a typing mechanism inspired by the PASCAL programming language, with added functions to provide mappings between abstract values and underlying bit patterns.

STRICT supports the design of generic components, by allowing blocks to declare formal parameters in the block heading. The actual parameters, which have to be specified when a block is instantiated, are then used in the structure section by recursively instantiating the same component, but with a different actual parameter (usually a smaller one). As remarked above, this leads to very concise descriptions which are highly amenable to formal verification.

Since STRICT allows blocks to be imported from other designs or from design libraries, reuse of components is possible and straightforward.

12.3. Disadvantages

We now turn our attention to some of the drawbacks of the language, and the lessons that can be learned from them.

The hints used to generate acceptable layout (pin placement, 'place' statements, **collapse** keyword) look like a 'hack'. They were bolted on to the language as an afterthought, they clutter up design descriptions, and are too limited anyway. For example, an underlying assumption is that all layout shapes are rectangular, which is clearly not always the case.

The mechanism to select a module generator was also bolted on as an afterthought, and clearly looks like it.

The same applies to the 'control' statement. This again looks like a language in itself, and should as a minimum have been designed in a functional notation. It was added to the language because it was originally not realised that control could not be extracted from just the behaviour and the structure (the behaviour section describes a design as a 'black box', i.e. how the outputs are generated as a result of current inputs and current states. The interaction between internal components is not described, partly because we wanted to move away from structural descriptions altogether).

The use of special keywords to provide hints to individual design tools violates one of the requirements of the language. In retrospect, these keywords should clearly not have been made available, even though they are optional. It should be left to the design tools to sort

out their own problems – the design language should be kept as clean and tool independent as possible.

The timing model is the most limited part of the language. It does not allow absolute time measures, such as the VHDL constructs 'ps', 'ns', 'ms', etc. This can cause major problems when importing cells from libraries – it is not assured that these cells use exactly the same timescales (a possible source of serious design bugs). The 'within' statement is not accurate enough – while it is true that the exact moment at which switching takes place may vary, designers would normally want to model it accurately anyway, using conservatively chosen values. The 'set' statement does not allow a range of values to be assigned, which would be important in accurate modelling. To be fair, the work reported upon in this thesis was mainly aimed at the formal verification and synthesis aspects of the language, and we are quite happy with a timing model that is adequate for most purposes.

Our description of finite state machines (and sequential systems in general) is somewhat different from non-sequential ones. In fact, it looks like a separate language altogether. This is the result of compromises made during (often heated) discussions with the electrical engineers in the group, who were unhappy at the thought of simulating a novel language which did not clearly spell out the states used during simulation. The language would clearly have benefited from a more uniform notation, in which function calls would have been semantically mapped onto states. Such an alternative notation would have made a description look more uniformly like a collection of function definitions and function calls. We believe this would now be more acceptable to electrical engineers.

An important cornerstone of the STRICT philosophy is that behavioural specification is mandatory. However, since the layout interface ignores the behavioural section completely, it is quite possible for a designer to insert a syntactically correct dummy behaviour such as

```
whenever rise(clock) =>  
    within (1)  
        set out=in;
```

and go ahead with old style structural design methods. This problem is largely unavoidable, and is reminiscent of the old saying that 'a determined programmer can write FORTRAN in any programming language'; this applies to chip designers as well! Only when it comes to using advanced behavioural tools such as the transformer, it is no longer possible to avoid writing proper behavioural specifications.

A number of desirable features have not ended up in the language. In particular, the functional expressions in the behavioural sections are rather limited; they do not support lists and higher order functions. Integers are limited in size to those of the host machine (i.e. multi-precision arithmetic is not provided).

A minor irritation is that the syntax of the language could have been simpler, thereby making descriptions even more concise. It was necessary to insert various punctuation marks in several places, because without them the grammar was not acceptable to the parser generator (the LALR requirement was violated). For example, the type definition

```
type  y ::= {is [1..5]}
```

would have been much preferable in the form

```
type  y ::= is 1..5
```

but the parser generator simply would not allow this.

In addition, the 'make' statement is really superfluous. Connections to the outside world could just as easily be specified in the 'using' statement.

STRICT has as yet no formally defined semantics. We clearly felt this was outside the scope of this thesis, and there were no other members of the (small) group inclined to take up this issue.

We believe that STRICT would benefit from a larger collection of built-in types and functions, such as the Verilog ones that detect particular clock edges. This would clearly help in making STRICT descriptions yet more concise and readable, and would increase simulation efficiency.

We want to mention the fact that the language was evaluated, along with a number of other languages and formalisms (which included ELLA, LTS, Pascal, Occam, VDM and CCS),

as part of the Alvey CAD002 project [1]. A substantial example, a Pythagoras DSP processor, was coded in STRICT as part of the evaluation. The conclusion was, not surprisingly, that none of the formalisms investigated was perfect, due to the lack of high quality design tools and the difficulty of handling timing aspects, but that the use of behavioural languages, including STRICT, represented a significant step forward towards the aim of the correct design of large VLSI systems. The report also notes about the use of recursion: "Plessey Caswell engineers found this concept difficult. It is not clear whether this is a fundamental problem or a longer training period is necessary". Perhaps the problem will diminish when a new generation of 'high level' designers take up jobs in industry.

As far as the design system is concerned, we want to make the following points:

- Having a syntax directed editor as the front end of a design system is extremely useful. It prevents the user from making errors, and is able to give hints as to the correct syntax of language statements. From the point of view of the programmer, it is very helpful to be able to use the generated syntax tree. The programmer just needs to write a set of recursive descent routines to extract the information from the parse tree, which is a straightforward task. This produces a large but reliable software module.
- The recursive descent module forms part of the set of procedural interface routines. Procedural interfaces are a popular concept in design systems where a large number of different tools must be interfaced to a language description. We have shown that we can interface a functional language to a large variety of tools using a procedural interface that extensively uses recursive programming techniques. This was most dramatically demonstrated with the Boyer–Moore prover, which was discovered as a possible tool well after the language had been defined. The Boyer–Moore interface clearly shows the effectiveness of the strategy used to interface to other systems.
- The recursive structure of the language also played an important part in the design of the builder module. Recursive programming techniques were used to match the recursive structures in the language. This allowed the code to be developed quickly and concisely, and (we believe) relatively free of bugs. We would regard its recursive

structure as an excellent example of how to deal with the problem in hand. The use of recursion, even in procedural programming languages, is obviously an aid to correct software development. The builder has been extensively used by students, and very few bugs have come to light.

A number of useful features are missing from the design system:

- There are no module generators for RAM and ROM. Unfortunately, the CADENCE version installed at Newcastle does not support these either, nor does it have a PLA generator.
- The design system does not support test generation or fault simulation. These however, are supported by CADENCE.
- Every VLSI design system should interface to a database management/version control system because of the occurrence of multiple versions and representations of designs at various levels. This has not been done in the STRICT system.
- If the whole system were to be rewritten from scratch, it would certainly be written in an object oriented language. This would vastly increase the ease with which new interfaces could be added.

12.4. Practical experience

Most of the comments made above about the language and design system have been confirmed by users. The language and system have been used for a number of years to teach VLSI design to postgraduate students. The main conclusions drawn from this experience are as follows:

- students have no problems with the structural part of the language.
- students do have problems with the behavioural part of the language, since the functional style forces them to express the behaviour in a certain manner. This makes them think hard about the specification, which must be regarded as a positive point.
- the viewer is the most popular part of the design system, and really seems to help the structural design effort quite a lot.

- few bugs have appeared during use of the system; it appears to be quite stable. This really seems to be a result of the recursive techniques used to implement data structures and functions. Again, the use of a functional notation appears to have been very helpful here.

12.5. Final conclusions and future work

As far as the timeliness of the STRICT language is concerned, we have shown that STRICT has some advantages over other languages, in particular the use of recursive functions as a central feature of the language. This is a feature absent from other languages, or present only in a limited form. The language could be improved in a number of areas. This is a matter of ongoing research.

13. REFERENCES

- [1] The Alvey Directorate, "Behavioural languages for VLSI", Final Report, Project CAD002, Ref. ALV/APP/CAD/002, 1986.
- [2] Balakrishnan, M., and Marwedel, P., "Integrated Scheduling and Binding: A Synthesis Approach for Design Space Exploration", Proc. 26th ACM/IEEE Design Automation Conference, pp. 68–74, 1989.
- [3] Backus, J. "Can programming be liberated from the Von Neumann style? A functional style plus its algebra of programs", CACM, Vol. 21, No. 8, pp. 613–641, Aug. 1978.
- [4] Barbacci, M.R., "Instruction Set Processor Specifications (ISPS): The Notation and Its Applications", IEEE Trans. Computers, vol. C-30, pp. 24–40, 1981.
- [5] Constable, R.L., "Implementing mathematics with the Nuprl Proof Development System", Prentice-Hall, 1986.
- [6] Barth, R., Solet, B., and Snidhu, P., "Parameterized schematics", 25th ACM/IEEE Design Automation Conference Proceedings, pp. 243–249, 1988.
- [7] Boyer, R.S., and Moore, J.S., "Proving Theorems About LISP Functions", Journal of the Association for Computing Machinery, vol. 22, no. 1, pp. 129–144, 1975.
- [8] Boute, R.T., "Current work on the Semantics of Digital Systems", Proc. 1985 workshop on VLSI, Edinburgh, "Formal Aspects of VLSI Design", Elsevier, pp. 99–112, 1986.
- [9] Boyer, R.S., and Moore, J.S., "A Computational Logic Handbook", Academic Press, Boston, 1988.
- [10] Brayton, R.K., Camposano, R., DeMicheli, G., Otten, R.H, vanEijndhoven, J. "The Yorktown Silicon Compiler". In: "Silicon Compilation", Gajski, (Ed), Addison Wesley, pp. 122–152, 1988.
- [11] Brock, B.C., and Hunt, W.A., "A Formal Introduction to a simple HDL", Technical Report 60, CLI Inc., Austin, 1990.

- [12] Buset, O.A., and Elmasry, M.I., "ACE: A Hierarchical Graphical Interface for Architectural Synthesis", Proc. 26th ACM/IEEE Design Automation Conference, pp. 537–542, 1989.
- [13] Cardelli, L., "An Algebraic approach to hardware description and verification", PhD thesis, University of Edinburgh, 1982.
- [14] Camposano, R., "Behaviour-Preserving Transformations for High-Level Synthesis", LNCS 408, Springer, pp. 106–128, 1990.
- [15] Campbell, R.H., and Richards, P.G., "SAGA: a system to automate the management of software production", AFIPS 50, pp. 231–234, 1981.
- [16] Camposano, R., and Rosentiel, W., "Synthesizing Circuits From Behavioural Descriptions", IEEE Trans on CAD, vol. CAD–8, no. 2, pp. 171–180, 1989.
- [17] Camposano, R., "Structural Synthesis in the Yorktown Silicon Compiler", Proc. VLSI–87, North Holland, pp. 61–72, 1987.
- [18] Camposano, R., and Tabet, R.M., "Design Representation for the Synthesis of Behavioural VHDL Models", Proc. IFIP International Working Conference, "Hardware Description Languages & their Applications", pp. 49–58, 1990.
- [19] Clark, W., "The Gantt chart." 3rd ed., Pitman and sons, London, 1952.
- [20] Cohn, A. "Correctness properties of the Viper block Model: The Second Level", Technical Report 134, Computer Laboratory, University of Cambridge, 1988.
- [21] DeMan, H., Rabaey, J., Six, P., and Claesen, L., Cathedral II: A Silicon Compiler for Digital Signal Processing", IEEE Design and Test 3, 6, pp. 13–25, 1986.
- [22] Denyer, P.B., "SAGE: A Methodology and Toolset for Architectural Synthesis", Technical Report SARI–035–B, Department of Electrical Engineering. Edinburgh University, 1988.
- [23] Dillinger, T.E., McCarthy, K.M., Masher, T.A., Neumann, D.R., and Schmidt, R.A., "A Logic Synthesis System for VHDL Design Descriptions", Proc. ICCAD–89, pp. 66–69, 1989.

- [24] EDIF Steering Committee, "EDIF Electronic Design Interchange Format Version 2.0.0", Electronic Industries Association, 1987.
- [25] Elmasry, M.I., Buset, O.A. "ACE, A Hierarchical Graphical Interface for Architectural Synthesis", Proc. 26th ACM/IEEE Design Aut. Conf., pp. 378–381, 1989.
- [26] Eveking, H., "Verification, Synthesis and Correctness–Preserving Transformations – Co-operative Approaches to correct Hardware design", Proc. IFIP International working conference, "From HDL descriptions to guaranteed correct circuit designs", Grenoble, pp. 229–239, 1986.
- [27] Finn, S., Fourman, M.P., Francis, M., and Harris, R., "Formal System Design – Interactive Synthesis based on Computer–Assisted Formal Reasoning", Proc. IFIP Workshop, "Applied Formal Methods for Correct VLSI Design", Belgium, pp. 97–110, 1989.
- [28] Ghezzi, C. and Mandroli, D. "Incremental Parsing", ACM TOPLAS, Vol. 1, pp. 58–70, 1979.
- [29] Gordon, M.J., "Register Transfer Systems and Their Behaviour", Proc. 5th Int. Conf. on Hardware Description Languages, pp. 88–93, 1981.
- [30] Gordon, M., "Proving a Computer Correct", Technical Report 42, University of Cambridge, Computer Laboratory, 1983.
- [31] Gordon, M., "Hardware Verification using Higher Order Logic", Technical Report 91, University of Cambridge, Computer Laboratory, 1986.
- [32] Gordon, M., "Why higher–order logic is a good formalism for specifying and verifying hardware", Proc. 1985 workshop on VLSI, Edinburgh, "Formal Aspects of VLSI Design", Elsevier, pp. 153–157, 1986.
- [33] Gordon, M., "HOL: A Proof Generating System for Higher–Order Logic", Technical Report 103, University of Cambridge, Computer Laboratory, 1987.
- [34] Gray, J.P., Buchanan, I., Robertson, P.S., "Designing Gate Arrays using a Silicon Compiler", Proc. 19th Design Aut. Conf., pp. 178–183, 1982.

- [35] Hanna, F.K., and Daeche, N., "Specification and Verification of digital systems using higher-order predicate logic", IEE Proc, vol. 133, Pt. E, no. 5, pp. 242–254, 1986.
- [36] Hanna, F.K., and Daeche, N., "Specification and Verification using Higher-Order Logic: A Case Study", Proc. 1985 workshop on VLSI, Edinburgh, "Formal Aspects of VLSI Design", Elsevier, pp. 179–213, 1986.
- [37] Hanna, F.K., Daeche, N., and Longley, M., "VERITAS: A Specification Language based on Type Theory", Proc. Cornell Mathematical Sciences Institute Workshop, "Hardware Specification, Verification and Synthesis", pp. 358–379, 1989.
- [38] Hartley, R.I., and Jasica, J.R., "Behavioural to Structural Translation in a Bit-Serial Silicon Compiler", IEEE Trans on CAD, vol. CAD-7, no. 8, pp. 877–885, 1988.
- [39] Hoshino, T. "UDL/I Version Two: A new horizon of HDL standards", Proc. CHDL '93, Ottawa, North Holland, pp. 437–452, 1993.
- [40] Hunt, W.A., "FM8501: A Verified Microprocessor", Ph.D. Thesis, University of Texas at Austin, December, 1985.
- [41] IEEE STD 1076–1987, "IEEE Standard VHDL Language Reference Manual", IEEE, New York, 1988.
- [42] Kalker, T., HOL Semantics for DSP, Philips Research Labs Technical Report, Eindhoven, The Netherlands, 1988.
- [43] Kramer, H., and Rosentiel, W., "System Synthesis using Behavioural Descriptions", Proc. EDAC-90, pp. 277–282, 1990.
- [44] Lanneer, D., Catthoor, F., Goosens, G., Pauwels, M., Meerbergen, J.V., and Man, H.D., "Open-ended System for High-Level synthesis of Flexible Signal Processors", Proc. EDAC-90, pp. 272–276, 1990.
- [45] Liesenberg, H., and Kinniment, D.J., "Placement expanding autolayout router", IEE Proc., Vol 133, Pt. I, No. 2, pp. 55–60, April 1986.

- [46] Locanthi, B., LAP: A SIMULA Package for I.C. Layout, Cal. Tech. Display File No. 1862, 1978.
- [47] May, D., and Keane, C., "Compiling OCCAM into Silicon", Communicat-ing Process Architecture Document, Meiko, April 1986.
- [48] McFarland, M.C., "Using Bottom-Up Techniques in the Synthesis of Digi-tal Hardware from Abstract Behavioural Descriptions", Proc. 23rd ACM/IEEE Design Automation Conference, pp. 474-479, 1986.
- [49] McFarland, M.C., Parker, A.C., and Camposano, R., "Tutorial on High-Level Synthesis", Proc. 25th ACM/IEEE Design Automation Conference, pp. 330-336, 1988.
- [50] McFarland, M.C., "Formal Verification of Sequential Hardware: A Tu-torial", IEEE Trans. CAD, Vol. 12, No. 5, pp. 633-653, May 1993.
- [51] Mead, C. and Conway, L., "Introduction to VLSI Systems", Addison-Wesley, 1980, pp. 115-127.
- [52] Morison, D., Peeling, N.E., and Thorp, T.L., "The design rationale of ELLA, a hardware design and description language", Proc. Computer Hardware descrip-tion languages and their applications, Tokyo, pp. 303-320, 1985.
- [53] Moszkowski, B., "A Temporal Logic for Multilevel Reasoning about Hard-ware", IEEE Computer, Vol.18 No.2, pp. 10-19, 1985.
- [54] Mukherjee, A., "Introduction to nMOS and CMOS VLSI Systems Design", Prentice Hall, 1986.
- [55] Peng, Z., Kuchinski, K., and Lyles, B., "CAMAD: A Unified Data Path / Control Synthesis Environment", IFIP Conf. "Design Methodologies for VLSI and Com-puter Architectures", pp. 53-67, 1989.
- [56] Pierre, L., "EECAP: A simple method to generalize the proof of equalities involving recursive functions with an accumulating parameter", Rapport de recherche MAIUP no. 89-09, Universite de Provence, Marseille, France, 1989.

- [57] Pierre, L., "The Formal Proof of Sequential Circuits described in CASCADE using the Boyer–Moore Theorem Prover", Proc. IFIP Workshop, Applied Formal Methods for Correct VLSI Design", Belgium, pp. 365–384, 1989.
- [58] Pierre, L., "The Formal Proof of the "Min–max" sequential benchmark described in CASCADE using the Boyer–Moore Theorem Prover", Proc. IFIP Workshop, "Applied Formal Methods for Correct VLSI Design", Belgium, pp. 129–148, 1989.
- [59] Rammig, F.J., "Modelling and simulation concepts of DACAPO II", Dosis GmbH, 1986.
- [60] Rushby, J., von Henke, F., and Owre, S., "An introduction to formal specification and verification using EHDM", Technical Report SRI-CSL–91–4, Computer Science Laboratory, SRI International, January 1991.
- [61] Scheichenzuber, J., Grass, W., Lauther, U., and Marz, S., "Global Hardware Synthesis from Behavioural Dataflow Descriptions", Proc. 27th ACM/IEEE Design Automation Conference, pp. 456–461, 1990.
- [62] Sheeran, M., "muFP, a Language for VLSI design", Proc. ACM Symp. on LISP and Functional Programming, pp. 104–112, 1984.
- [63] Siskind, J.M., Southard, J.R. and Grouch, K.W., "High–performance VLSI designs from Succinct Algorithmic descriptions", Proc. Conf. on advanced research in VLSI, MIT, pp. 69–79, 1982.
- [64] SPICE, "SPICE 3f2, Users Manual", Univ. of California at Berkeley, 1992.
- [65] Suzuki, N., "Concurrent Prolog as an efficient VLSI Design Language", IEEE Computer, Vol. 18, No. 2, pp. 33–40, 1985.
- [66] Verkest, D., Johannes, P., Claesen, L., De Man, H. "Correctness proofs of parameterized hardware modules in the Cathedral–II synthesis environment". In Proc. EDAC–90, pp. 62–66, 1990.
- [67] Verkest, D., Claesen, L., and Man, H.D., "On the use of the Boyer–Moore theorem prover for correctness proofs of parameterized hardware modules", Proc. IFIP

Workshop, "Applied Formal Methods for Correct VLSI Design", Belgium, pp. 405–422, 1989.

[68] Walker, R.A., and Thomas, D.E., "Behavioural transformation for Algorithmic level IC Design", IEEE Trans. CAD, pp. 1115–1128, Vol. 8, No. 10, 1989.

[69] Zegers, J., Six, P., Rabaey, J., and Man, H.D., "CGE: Automatic Generation of Controllers in the CATHEDRAL–II Silicon Compiler", Proc. EDAC–90, pp. 617–621, 1990.

[70] Megson, G.M., "Sorting without exchanges on a bit–serial systolic array", IEE Proc., Vol. 137, Pt. G., No. 5, pp. 345–352, 1990.

[71] Robson, A.P., "SIMSTRICT, a behavioural simulator for use with the STRICT Hardware Description Language", Technical Report, Dept. of Electrical and Electronic Engineering, Univ. of Newcastle upon Tyne, 1989.

[72] Thomas, D.E., and Moorby, P., "The Verilog Hardware Description Language", Kluwer Publishers, 1991.

[73] Aylor, J.H., Waxman, R., and Scarratt, C., "VHDL – Feature Description and Analysis", IEEE Design and Test of Computers, pp. 17–27, April 1986.

[74] "Verilog–XL: Product Description", Gateway Design Automation Corporation, 1989.

[75] Kloos, C.D., "Formal Semantics for VHDL". Kluwer Publishers, 1995.

[76] Goossens, K.G.W., "Semantics for picoELLA", Technical report, Dept. of Computer Science, University of Edinburgh, 1990.

[77] Cohn, A., "Correctness properties of the VIPER block Model: The Second Level", in: Current Trends in Hardware Verification and Automatic Theorem Proving, Springer, pp. 1–91, 1989.

[78] Borriane, D., Pierre, L., and Salem, A., "PREVAIL: A Proof Environment for VHDL Descriptions", in: Correct Hardware Design Methodologies, Elsevier, pp. 163–186, 1992.

- [79] Turner, D., "Recursion equations as a programming language" In: "Functional programming and its applications", Cambridge University Press, pp. 1–28, 1982.
- [80] Henderson, P., "Functional Programming: Application and Implementation", Prentice–Hall International, Series in Computer Science, 1980.

BIBLIOGRAPHY

The papers below, whilst not directly referenced in this thesis, give a good overview of the general research climate in which the work was performed.

- [81] Arndt, R.L., and Dietmeyer, D.L., "DDLSIM—A Digital Design Language Simulator", Proc. NEC, pp. 116–118, 1970.
- [82] Berman, C.L., and Trevillyan, L.H., "Functional Comparison of Logic Designs for VLSI Circuits", Proc. ICCAD–89, pp. 456–459, 1989.
- [83] Bhasker, J., "Process–Graph Analyser: A Front–End Tool for VHDL Behavioural Synthesis", SOFTWARE–PRACTICE AND EXPERIENCE, VOL. 18(5), pp. 469–483, 1988.
- [84] Birtwistle, G., Graham, B., Simpson, T., Slind, K., Williams, M., and Williams, S., "Verifying an SECD chip in HOL", Proc. IFIP Workshop, "Applied Formal Methods for Correct VLSI Design", Belgium, pp.149–158, 1989.
- [85] Blackman, T., Fox, J., and Rosebrugh, C., "The SILCtm SILICON COMPILER: Language and Features", Proc. 22nd ACM/IEEE Design Automation Conference, pp. 232–237, 1985.
- [86] Blackburn, R.L., Thomas, D.E., and Koenig, P.M., "CORAL II: Linking Behaviour and Structure in an I.C. Design System", Proc. 25th ACM/IEEE Design Automation Conference, pp. 529–535, 1988.
- [87] Boyd, D.R.S., "APECS: A Pascal Environment for Circuit Specification", Internal Memo, Rutherford Appleton Laboratories, Chilton, Didcot, 1981.
- [88] Borrione, D., Paillet, J.L., and Pierre, L., "Formal Verification of CASCADE descriptions", Proc. IFIP International Working Conference, "The Fusion of Hardware Design and Verification", U.K., pp. 185–210, 1988.

- [89] Brown, G.M., and Leeson, M.E., "Synthesizing Correct Sequential Circuits", Proc. IFIP International Working Conference, "Computer Hardware Descriptions & their Applications," pp. 169–181, 1990.
- [90] Burns, F.P., Kinniment, D.J., and Koelmans, A.M., "Correct Interactive Transformational Synthesis of DSP Hardware", Proc. European Design Automation Conference, pp. 16–21, 1991.
- [91] Camposano, R., "Synthesis techniques for Digital Systems Design", Proc. 22nd ACM/IEEE Design Automation Conference, pp. 475–480, 1985.
- [92] Camposano, R., "Design Process Model in the Yorktown Silicon Compiler", Proc. 25th ACM/IEEE Design Automation Conference", pp. 489–494, 1988.
- [93] Camilleri, A.J., "Simulation as an aid to Verification using the HOL Theorem Prover", IFIP working conference, "Design Methodologies for VLSI and Computer Architectures", pp. 147–167, 1989.
- [94] Camurati, P., and Prinetto, P., "Formal Verification of Hardware Correctness: Introduction and Survey of Current Research", IEEE COMPUTER, pp. 8–19, 1988.
- [95] Cadence, "Verilog–XL, Reference Manual", Volumes 1 and 2, 1991.
- [96] Chu, C.M., Potkonjak, M., Thaler, M., and Rabaey, J., "HYPER: An Interactive Synthesis Environment for High Performance Real Time Applications", IEEE Design and Test of Computers, pp. 432–435, 1989.
- [97] Collavizza, H., "Functional Semantics of Microprocessors at the Microprogram Level and Correspondence with the Machine Instruction Level", Proc. EDAC–90, pp. 52–56, 1990.
- [98] Director, S.W., Parker, A.C., Siewiorek, D.P., and Thomas, D.E., "A Design Methodology and Computer Aids for Digital VLSI Systems", IEEE Trans. Circuits and Systems, vol. CAS–28, pp. 634–644, 1981.

- [99] Dussault, J., Liaw, C.C., and Tong, M.M., "A High Level Synthesis Tool for MOS Chip Design", Proc. 21st ACM/IEEE Design Automation Conference, pp. 308–313, 1984.
- [100] Dutt, N.D., and Gajski, D.D., "Designer Controlled Behavioural Synthesis", Proc. 26th ACM/IEEE Design Automation Conference, pp. 754–757, 1989.
- [101] Dutt, N.D., and Gajski, D.D., EXEL: A Language for Interactive Behavioural Synthesis", Proc. IFIP International Working Conference, "Computer Hardware Description Languages & their Applications", pp. 3–17, 1990.
- [102] Feldbusch, F., and Kumar, R., "Verification of Synthesized Circuits at Register Transfer Level with Flow Graphs", pp. 22–26, 1991.
- [103] Gajski, D.D., "Silicon Compilers and Expert Systems", Proc. 21st ACM/IEEE Design Automation Conference, pp. 86–87, 1984.
- [104] Gaboury, P., and Elmasry, M.I., "Using Program Transformation for VLSI Design Automation", Proc. IFIP Workshop, "Applied Formal Methods for Correct VLSI Design", Belgium, pp. 40–56, 1989.
- [105] Ghosh, A., Devadas, S., and Newton, A.R., "Verification of Interacting Sequential Circuits", Proc. 27th ACM/IEEE Design Automation Conference, pp. 213–219, 1990.
- [106] Gopalakrishnan, G.C., Smith, D.R., Shrivastava, M.K., "An algebraic approach to the Specification and Realization of VLSI designs", In: "Computer Hardware Description Languages and their applications", North-Holland, pp. 16–38, 1985.
- [107] Haroun, B.S., and Elmasry, M.I., "SPAID: An Architectural Synthesis Tool for DSP Custom Applications", IEEE 1988 Custom Integrated Circuits Conference, pp. 14.4.1–14.4.5, 1988.
- [108] Halbwachs, N., Louchemy, A., and Pilaud, D., "Describing and designing circuits by means of a synchronous declarative language", IFIP International working conference, "From HDL descriptions to guaranteed correct circuit designs", Grenoble, pp. 255–268, 1986.

- [109] Hanna, F.K., Longley, M., and Daeche, N., "Formal Synthesis of Digital Systems", Proc. IFIP Workshop, "Applied Formal Methods for Correct VLSI Design", Belgium, pp. 532–548, 1989.
- [110] Hilfinger, P.N., "A High-Level Language and Silicon Compiler for Digital Signal Processing", IEEE 1985 Custom Integrated Circuits Conf., pp.213–216, 1985.
- [111] Hodgson, S. "A Multilevel, Mixed State Simulator for Hierarchical Design Verification", Proc. IEEE Electronic Design Automation, pp. 107–110, 1984.
- [112] Hunt, W.A., "Microprocessor Design Verification", Technical Report 48, CLI Inc., Austin, 1989.
- [113] Jerraya, A., Varinot, P., Jamier, R., and Courtois, B., "Principles of the Syco Compiler", Proc. 23rd ACM/IEEE Design Automation Conference, pp. 715–721, 1986.
- [114] Joepen, H., and Glesner, M., "Optimal Structuring of Hierarchical Control-Paths in a Silicon-Compiler System", Proc. ICCAD–86, pp. 264–267, 1986.
- [115] Johnson, S.D., "Synthesis of Digital Designs from Recursion Equations", ACM Distinguished Dissertation, MIT Press, 1983.
- [116] Johnson, S.D., "Digital Design in a Functional Calculus", Proc. 1985 workshop on VLSI, Edinburgh, "Formal Aspects of VLSI Design", Elsevier Science Publishers, B.V. (North-Holland), pp. 45–57, 1986.
- [117] Johannsen, D., "Bristle Blocks: a Silicon Compiler", Proc. 16th Design Automation Conf., pp. 310–313, 1979.
- [118] Johnson, S.D., Wehrmeister, R.M., and Bose, B., "On the Interplay of Synthesis and Verification Experiments with the FM8501 Processor Description", Proc. IFIP Workshop, Applied Formal Methods for Correct VLSI Design", Belgium, pp. 385–403, 1989.
- [119] Kalker, T., "Formal Methods for Silicon Compilation", Proc. European Design Automation Conference, pp. 395–400, 1991.

- [120] Kruatrachue, B., and Lewis, E., "Grain size determination for parallel processing", *IEEE Software*, vol. 5, No 1, pp 23–31, 1988.
- [121] Losleben, P. "Computer Aided Design for VLSI" In: "Very Large Scale Integration (VLSI) Fundamentals and Applications", Springer, pp. 89–127, 1982.
- [122] Luk, W., and Jones, G., "From Specification to Parameterized Architectures", *Proc. IFIP International Working Conference, "The Fusion of Hardware Design and Verification"*, pp. 267–288, 1988.
- [123] Martin, A.J., "A synthesis Method for Self-timed VLSI circuits", *Proc. ICCD 87, International Conference on Computer Design*, pp. 224–229, 1987.
- [124] Madre, J.C., and Billon, J.P., "Proving Circuit Correctness using Formal Comparison Between Expected and Extracted Behaviour", *Proc. 25th ACM/IEEE Design Automation Conference*, pp. 205–210, 1988.
- [125] De Man, J., "Transformational Design: A Case Study", *Proc. IFIP Workshop, "Applied Formal Methods for Correct VLSI Design"*, Belgium, pp. 206–215, 1989.
- [126] McFarland, M.C., and Parker, A.C., "An Abstract Model of Behaviour for Hardware Descriptions", *IEEE Trans. Computers*, vol. C-32, pp. 621–637, 1983.
- [127] Meshkinpour, F., and Ercegovic, M.D., "A Functional Language for Description and Design of Digital Systems: Sequential Constructs", *Proc. 22nd ACM/IEEE Design Automation Conference*, pp. 238–244, 1985.
- [128] Milne, G.J., "Behavioural description and VLSI verification", *IEE Proc.*, vol. 133, Pt E, no. 3, pp. 127–137, 1986.
- [129] Nash, J.H. and Smith, S.G., "A Front End Graphics Interface To The First Silicon Compiler", *Proc. IEEE Electronic Design Automation*, pp. 120–124, 1984.
- [130] Narendran, p., and Stillman, J., "Formal Verification of the Sobel Image Processing Chip", *Proc. 25th ACM/IEEE Design Automation Conference*, pp. 211–217, 1988.


- [131] O'Donnell, J.T., "Hydra: Hardware description in a functional language using recursion equations and higher order combining forms", Proc. IFIP Conf. "The Fusion of Hardware Design and Verification", pp. 309–328, 1988.
- [132] Orailoglu, A., and Gajski, D.D., "Flow Graph Representation", Proc. 23rd ACM/IEEE Design Automation Conference, pp. 503–509, 1986.
- [133] Paillet, J., "A Functional Model for Descriptions and Specifications of Digital Devices", Proc. IFIP International working conference", From HDL descriptions to guaranteed correct circuit designs", Grenoble, pp. 21–42, 1986.
- [134Pe86] Peng, Z., "Synthesis of VLSI Systems with the CAMAD Design Aid", Proc. 23rd ACM/IEEE Design Automation Conference, pp. 278–283, 1986.
- [135] Piloty, R., Borrione, D., "The CONLAN Project: Concepts, Implementations, and Applications", IEEE Computer, pp. 81–92, Feb. 1985.
- [136] Razouk, R.R., "The Use of Petri Nets for Modeling Pipelined Processors", Proc. 25th ACM/IEEE Design Automation Conference, pp. 548–553, 1988.
- [137] Rajopadhye, S.V., "Algebraic Transformations in Systolic Array Synthesis: A Case Study", Proc. IFIP Workshop, "Applied Formal Methods for Correct VLSI Design", Belgium, pp. 281–290, 1989.
- [138] Russel, G., Kinniment, D.J., Chester, E.G., and McLauchlan, M.R., "CAD for VLSI", Van Nostrand Reinhold, U.K., 1985.
- [139] Sarma, R.C., Dooley, M.D., Newman, N.C., and Hetherington, G., "High-Level Synthesis: Technology Transfer To Industry", Proc. 27th ACM/IEEE Design Automation Conference, pp. 549–554, 1990.
- [140] Southard, J.R., "MacPitts: An Approach to Silicon Compilation", IEEE COMPUTER, pp. 74–82, 1983.
- [141] Spreitzer, M., "Comparing Structurally Different Views of a VLSI Design", Proc. 27th ACM/IEEE Design Automation Conference, pp. 200–206, 1990.

- [142] Stavridou, V., Barringer, H., and Edwards, D.A., "Formal Specification and verification of Hardware: A Comparative Case Study", Proc. 25th ACM/IEEE Design Automation Conference, pp. 197–203, 1988.
- [143] Tamassia, R., Di Battista, G., and Batini., C., "Automatic graph drawing and the readability of diagrams" IEEE Transactions on systems, man, and cybernetics, Vol 18, no 1, pp. 61–79, 1988.
- [144] Thomas, D.E., Dirkes, E.M., Walker, R.A., Rajan, J.V., Nestor, J.A., and Blackburn, R.L., "The System Architect's Workbench", Proc. 25th ACM/IEEE Design Automation Conference, pp. 337–343, 1988.
- [145] Trickey, H., "Flamel: A High-Level Hardware Compiler", IEEE Trans on CAD, vol. CAD-6, no. 2, pp. 259–269, 1987.
- [146] Tseng, C.J., Wei, R.S., Rothweiler, S.G., Tong, M.M., and Bose, A.K., "Bridge: A Versatile Behavioural Synthesis System", Proc. 25th ACM/IEEE Design Automation Conference, pp. 415–420, 1988.
- [147] Vanhoof, J., Rabaey, J., and Man, H.D., "A Knowledge-Based CAD System for Synthesis of Multi-Processor Digital Signal Processing Chips", Proc. IFIP International working conference, VLSI-87, pp. 73–88, 1987.
- [148] Vemuri, R., "A Formal Model for Register Transfer Level Structures and Its Applications in Verification and Synthesis", Proc. IFIP Workshop, "Applied Formal Methods for Correct VLSI Design", Belgium, pp. 77–96, 1989.
- [149] Walker, R.A., and Thomas, D.E., "Design Representation and Transformation in the System Architect's Workbench", Proc. ICCAD-87, pp. 166–169, 1987.
- [150] Whitcomb, G.S., and Newton, A.R., "Abstract Data Types and High-Level Synthesis", Proc. 27th ACM/IEEE Design Automation Conference, pp. 680–685, 1990.
- [151] Wilk, A., and Pnuelli, A., "Specification and Verification of VLSI Systems", Proc. ICCAD-89, pp. 460–463, 1989.

APPENDICES

A. SAMPLE OUTPUTS

A.1. Simulator

Shown below in Fig. 39 is a screen dump for the full adder example of section 3.9.3. It shows how the simulator prompts for input values for the various ports, traces the operation of the device, and then displays the expected outputs. On each output line, the simulator first prints the current time value in square brackets, followed by a code indicating the sub-systems involved in the output. The main prompt is the symbol '>>>'.


0] SIM01I SIMSTRICT release 1.2

Simulating : F

```
0] SIM00R>>> decompose
0] COM06I 1 blocks decomposed, 0 with compare
0] SIM00R>>> deposit :*
0] COM07P Give binary number ( max 1 bits ) for F:X
0] COM11R*** 1
0] COM07P Give binary number ( max 1 bits ) for F:CIN
0] COM11R*** 0
0] COM07P Give binary number ( max 1 bits ) for F:COUT
0] COM11R***
0] COM07P Give binary number ( max 1 bits ) for F:S
0] COM11R***
0] COM07P Give binary number ( max 1 bits ) for F:Y
0] COM11R*** 1
0] COM14I 3 ports selected for deposit
0] SIM00R>>> go
0] MAN02I TRACE KDv D-- F:HT:Y 1
0] MAN02I TRACE KDv D-- F:Y 1
0] MAN02I TRACE KDv D-- F:HB:Y 0
0] MAN02I TRACE KDv D-- F:CIN 0
0] MAN02I TRACE KDv D-- F:HT:X 1
0] MAN02I TRACE KDv D-- F:X 1
10] MAN02I TRACE KDv D-- F:O:IN ?
10] MAN02I TRACE KDv D-- F:HB:X 0
10] MAN02I TRACE KDv DgD F:HT:S 0
10] MAN02I TRACE KDv DgD F:HT:C 1
16] MAN03E Signal driving conflict on port F:COUT
16] MAN02I TRACE KDv DgD F:COUT ?
16] MAN02I TRACE KDv DgD F:O:OUT ?
20] MAN03E Signal driving conflict on port F:S
20] MAN02I TRACE KDv D-- F:O:IN 10
20] MAN02I TRACE KDv DgD F:S 0
20] MAN02I TRACE KDv DgD F:HB:S 0
20] MAN02I TRACE KDv DgD F:HB:C 0
26] MAN03E Signal driving conflict on port F:COUT
26] MAN02I TRACE KDv DgD F:COUT 1
26] MAN02I TRACE KDv DgD F:O:OUT 1
26] MAN61I ALL events complete
26] SIM00R>>> show block ports
26] COM28I block F
    level = 0 type = FUNCT number = 349 DECOMPOSED
    behaviour = FULL

    ports    type    --- target    value
    X         IN     -T- $PAD:X      1
    CIN       IN     -T- $PAD:CIN    0
    COUT      OUT    -T- $PAD:COUT   1
    S         OUT    -T- $PAD:S      0
    Y         IN     -T- $PAD:Y      1

26] SIM00R>>> █
```

Fig. 39. Simulator screen dump

A.2. Layout

This section shows a number of layout plots, produced from GAELIC files which were generated by the STRICT system.

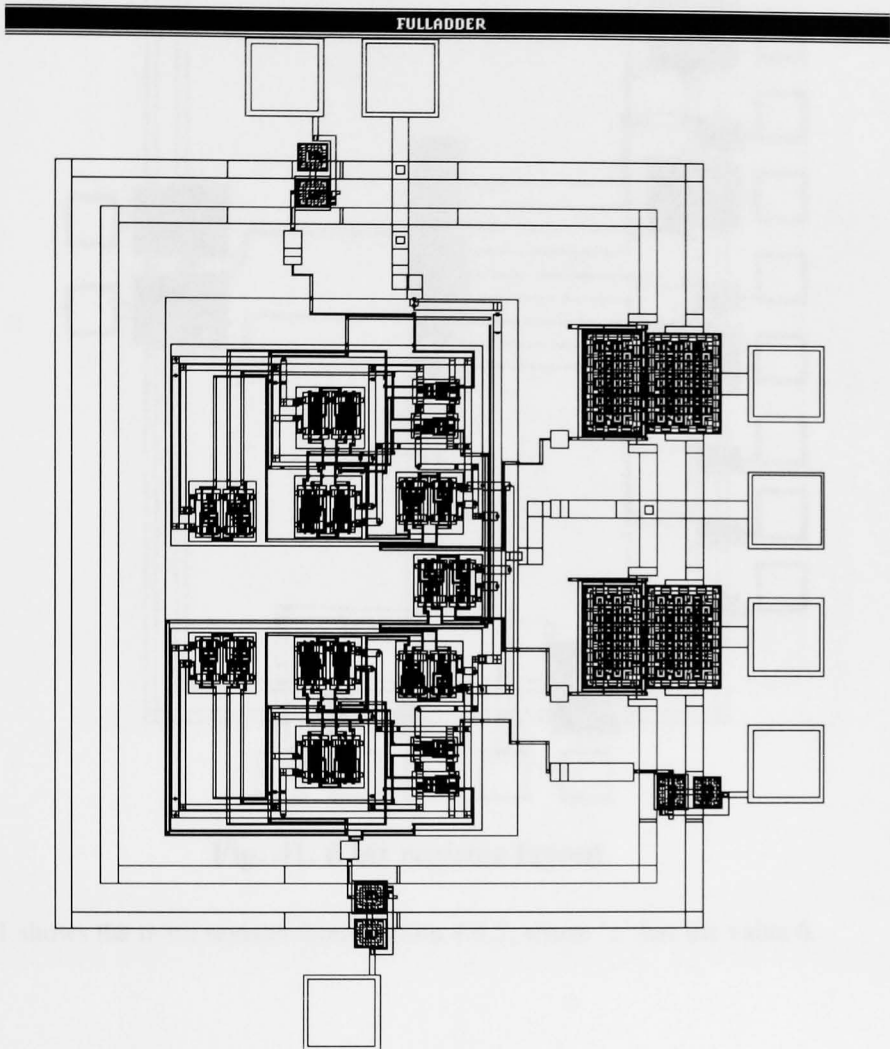


Fig. 40. Full adder layout

Figure 40 shows a plot for the full adder example of section 4.9.3.

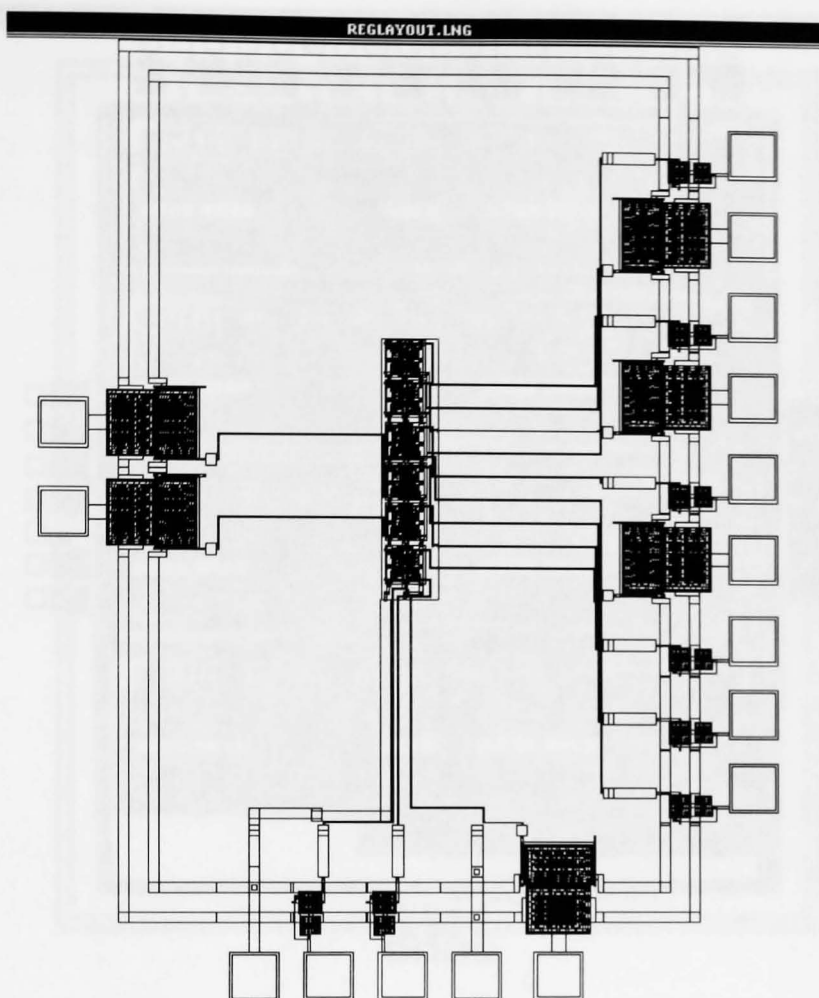


Fig. 41. 6 bit register layout

Figure 41 shows the n -bit register from section 4.9.5, where 'n' has the value 6.

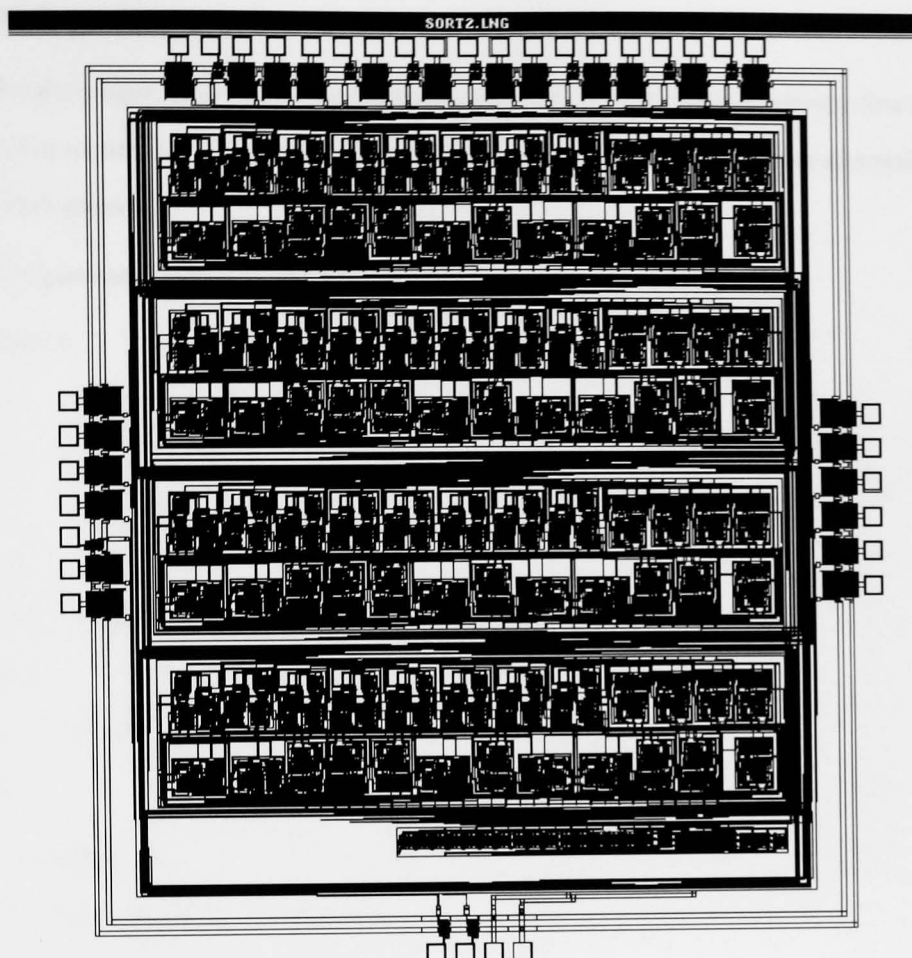


Fig. 42. Systolic array layout

Finally, Figure 42 shows a systolic array, similar to example 4.9.6. The actual design is Megson's systolic sorter [70], in which the basic cells contain flip flops and half adders.

A.3 PLA generator

The partial example shown below is the Mead and Conway traffic light controller first introduced in section 3.9.9. It shows clearly the kind of behavioural descriptions acceptable to the PLA generator.

The PLA generator generates the following STATIC code.

CONSTANTS

green = 0

yellow = 1

red = 2

INPUTS

carhere, longtimeout, shorttimeout

OUTPUTS

timer, road<2>, farmroad<2>

TABLE

STATE roadgreen

IF NOT carhere GOTO roadgreen

DO road = green, farmroad = red;

IF NOT longtimeout GOTO roadgreen

DO road = green, farmroad = red;

IF carhere AND longtimeout GOTO roadyellow

DO road = green, farmroad = red, timer = 1;

STATE roadyellow

IF NOT shorttimeout GOTO roadyellow

DO road = yellow, farmroad = red;

IF shorttimeout GOTO farmroadgreen

DO road = yellow ,farmroad = red, timer = 1;

STATE farmroadgreen

IF carhere AND NOT longtimeout GOTO farmroadgreen

DO road = red, farmroad = green;

IF NOT carhere GOTO farmroadyellow

```

DO road = red, farmroad = green, timer = 1;
IF longtimeou GOTO farmroadyellow
DO road = red, farmroad = green;

STATE farmroadyellow
IF NOT shorttimeout GOTO farmroadyellow
DO road = red, farmroad = yellow;
IFshorttimeout GOTO roadgreen
DO road = red, farmroad = yellow;

```

The STATIC parser translates this into a set of boolean equations, which are then minimised. The result is:

```

OUT1=IN2.FEED1.FEED0'+IN0.IN1'.FEED1'.FEED0+IN0'.FEED1'.FEED0+
      IN1.FEED1'.FEED0+IN2'.FEED1.FEED0 ;
OUT2=IN0'.FEED1'.FEED0+IN1.FEED1'.FEED0+IN0.IN1.FEED1'.FEED0'+
      IN2'.FEED1 ;
OUT3=IN2.FEED1.FEED0'+IN0'.FEED1'.FEED0+IN0.IN1.FEED1'.FEED0' ;
OUT4=IN0.IN1'.FEED1'.FEED0+IN0'.FEED1'.FEED0+IN1.FEED1'.FEED0+
      FEED1.FEED0 ;
OUT5=FEED1.FEED0' ;
OUT6=IN0.IN1.FEED1'.FEED0'+FEED1.FEED0'+IN0'.FEED1'.FEED0'+
      IN1'.FEED1'.FEED0' ;
OUT7=FEED1.FEED0 ;

```

Pla data after multiple output minimisation

Number of inputs	:	5
Number of outputs	:	7
Number of product terms	:	10

Extra output included to ensure even number

The minimised equations are translated into bitmaps for the AND and OR plane, as follows:

```

0 0
    5    8    2
    10
0 0    0 0    1 0    1 0    0 1
1 0    0 1    0 0    0 1    1 0
0 1    0 0    0 0    0 1    1 0
0 0    1 0    0 0    0 1    1 0
0 0    0 0    0 1    1 0    1 0
1 0    1 0    0 0    0 1    0 1
0 0    0 0    0 1    1 0    0 1
0 0    0 0    1 0    1 0    1 0
0 1    0 0    0 0    0 1    0 1
0 0    0 1    0 0    0 1    0 1

1 0 1 0 1 1 0 0
1 0 0 1 0 0 0 0
1 1 1 1 0 0 0 0
1 1 0 1 0 0 0 0
1 1 0 1 0 0 1 0
0 1 1 0 0 1 0 0
0 1 0 0 1 1 0 0
0 0 0 1 0 0 1 0
0 0 0 0 0 1 0 0
0 0 0 0 0 1 0 0

```

The bitmap is then translated into GAELIC. A plot of the result is shown in Fig. 43.

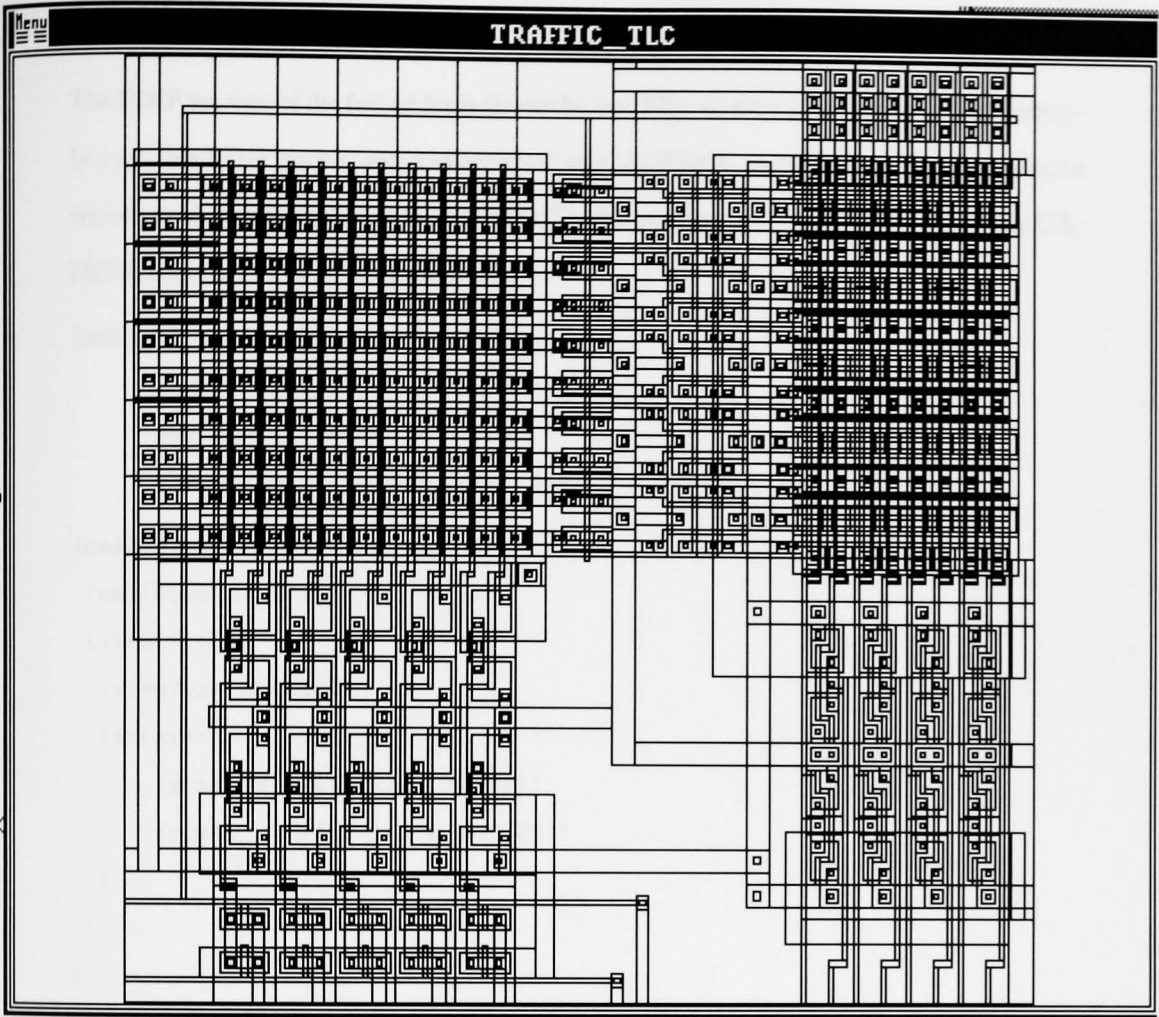


Fig. 43. PLA plot

The GAELIC file should then be merged with the output of the layout system. There is currently no EDIF interface available.

A.4. EDIF output

The EDIF version of the full adder is shown below. This version was checked with a publicly available EDIF parser, and was accepted by CADENCE as input. Unfortunately, due to serious problems with the CADENCE EDIF software it was not possible to include the CADENCE library cells in the description.

```
(edif CADENCE_EDIF
  (edifVersion 2 0 0)
  (edifLevel 0)
  (keywordMap (keywordLevel 0))

  (cell ntg
    (cellType GENERIC)
    (view v_ntg
      (viewType NETLIST)
      (interface
        (port a (direction INPUT))
        (port qb (direction OUTPUT))
      )
    )
  )

  (cell inverter
    (cellType GENERIC)
    (view v_inverter
      (viewType NETLIST)
      (interface
        (port a (direction INPUT))
        (port out (direction OUTPUT))
      )
    )

    (contents
      (instance x (viewRef v_ntg (cellRef ntg))
        (property placementStatus
          (string "suggested")
          (owner "Cadence")
        )
      )
    )
  )
)
```

```

    )
    (net N1
      (joined
        (portRef a )
        (portRef a (instanceRef x))
      )
    )
  )
  (net N2
    (joined
      (portRef qb (instanceRef x))
      (portRef out )
    )
  )
)

(cell nag2
  (cellType GENERIC)
  (view v_nag2
    (viewType NETLIST)
    (interface
      (port a (direction INPUT))
      (port b (direction INPUT))
      (port qb (direction OUTPUT))
    )
  )
)

(cell andgate2
  (cellType GENERIC)
  (view v_andgate2
    (viewType NETLIST)
    (interface
      (port a (direction INPUT))
      (port b (direction INPUT))
      (port out (direction OUTPUT))
    )
  )
)

```

```

)
(contents
  (instance x (viewRef v_nag2 (cellRef nag2))
    (property placementStatus
      (string "suggested")
      (owner "Cadence")
    )
  )
)
(instance y (viewRef v_ntg (cellRef ntg))
  (property placementStatus
    (string "suggested")
    (owner "Cadence")
  )
)
(net N1
  (joined
    (portRef a )
    (portRef a (instanceRef x))
  )
)
(net N2
  (joined
    (portRef b )
    (portRef b (instanceRef x))
  )
)
(net N3
  (joined
    (portRef qb (instanceRef x))
    (portRef a (instanceRef y))
  )
)
(net N4
  (joined
    (portRef qb (instanceRef y))

```

```

        (portRef out )
    )
)
)
)
)

(cell nor2
  (cellType GENERIC)
  (view v_nor2
    (viewType NETLIST)
    (interface
      (port a (direction INPUT))
      (port b (direction INPUT))
      (port qb (direction OUTPUT))
    )
  )
)

(cell orgate2
  (cellType GENERIC)
  (view v_orgate2
    (viewType NETLIST)
    (interface
      (port a (direction INPUT))
      (port b (direction INPUT))
      (port out (direction OUTPUT))
    )
  )
  (contents
    (instance x (viewRef v_nor2 (cellRef nor2))
      (property placementStatus
        (string "suggested")
        (owner "Cadence")
      )
    )
  )
  (instance y (viewRef v_ntg (cellRef ntg))
    (property placementStatus

```



```

        (string "suggested")
        (owner "Cadence")
    )
)
(net N1
  (joined
    (portRef a )
    (portRef a (instanceRef x))
  )
)
(net N2
  (joined
    (portRef b )
    (portRef b (instanceRef x))
  )
)
(net N3
  (joined
    (portRef qb (instanceRef x))
    (portRef a (instanceRef y))
  )
)
(net N4
  (joined
    (portRef qb (instanceRef y))
    (portRef out )
  )
)
)
)
)

(cell hal
  (cellType GENERIC)
  (view v_hal
    (viewType NETLIST)

```

```

(interface
  (port x (direction INPUT))
  (port y (direction INPUT))
  (port s (direction OUTPUT))
  (port c (direction OUTPUT))
)

(contents
  (instance n1 (viewRef v_inverter (cellRef inverter))
    (property placementStatus
      (string "suggested")
      (owner "Cadence")
    )
  )
)

(instance n2 (viewRef v_inverter (cellRef inverter))
  (property placementStatus
    (string "suggested")
    (owner "Cadence")
  )
)

(instance a1 (viewRef v_andgate2 (cellRef andgate2))
  (property placementStatus
    (string "suggested")
    (owner "Cadence")
  )
)

(instance a2 (viewRef v_andgate2 (cellRef andgate2))
  (property placementStatus
    (string "suggested")
    (owner "Cadence")
  )
)

(instance a3 (viewRef v_andgate2 (cellRef andgate2))
  (property placementStatus
    (string "suggested")
    (owner "Cadence")
  )
)

```

```

    )
  )
  (instance o1 (viewRef v_orgate2 (cellRef orgate2))
    (property placementStatus
      (string "suggested")
      (owner "Cadence")
    )
  )
)
(net N1
  (joined
    (portRef x )
    (portRef a (instanceRef a1))
    (portRef a (instanceRef a2))
    (portRef a (instanceRef n2))
  )
)
(net N2
  (joined
    (portRef y )
    (portRef b (instanceRef a1))
    (portRef b (instanceRef a3))
    (portRef a (instanceRef n1))
  )
)
(net N3
  (joined
    (portRef out (instanceRef n1))
    (portRef b (instanceRef a2))
  )
)
(net N4
  (joined
    (portRef out (instanceRef n2))
    (portRef a (instanceRef a3))
  )
)

```

```

)
(net N5
  (joined
    (portRef out (instanceRef a1))
    (portRef c )
  )
)
(net N6
  (joined
    (portRef out (instanceRef a2))
    (portRef a (instanceRef o1))
  )
)
(net N7
  (joined
    (portRef out (instanceRef a3))
    (portRef b (instanceRef o1))
  )
)
(net N8
  (joined
    (portRef out (instanceRef o1))
    (portRef s )
  )
)
)
)
)

(cell full
  (cellType GENERIC)
  (view v_full
    (viewType NETLIST)
    (interface
      (port x (direction INPUT))
      (port y (direction INPUT))
    )
  )
)

```

```

    (port cin (direction INPUT))
    (port cout (direction OUTPUT))
    (port s (direction OUTPUT))
)
(contents
  (instance ht (viewRef v_hal (cellRef hal))
    (property placementStatus
      (string "suggested")
      (owner "Cadence")
    )
  )
)
(instance hb (viewRef v_hal (cellRef hal))
  (property placementStatus
    (string "suggested")
    (owner "Cadence")
  )
)
(instance o (viewRef v_orgate2 (cellRef orgate2))
  (property placementStatus
    (string "suggested")
    (owner "Cadence")
  )
)
(net N1
  (joined
    (portRef x )
    (portRef x (instanceRef ht))
  )
)
(net N2
  (joined
    (portRef y )
    (portRef y (instanceRef ht))
  )
)

```

```

(net N3
  (joined
    (portRef cin )
    (portRef y (instanceRef hb))
  )
)
(net N4
  (joined
    (portRef s (instanceRef ht))
    (portRef x (instanceRef hb))
  )
)
(net N5
  (joined
    (portRef c (instanceRef ht))
    (portRef a (instanceRef o))
  )
)
(net N6
  (joined
    (portRef s (instanceRef hb))
    (portRef s )
  )
)
(net N7
  (joined
    (portRef c (instanceRef hb))
    (portRef b (instanceRef o))
  )
)
(net N8
  (joined
    (portRef out (instanceRef o))
    (portRef cout )
  )
)

```

)
)
)
)
)

A.5. Boyer–Moore Prover output

This appendix shows part of the output generated by the Boyer–Moore prover when presented with the input file shown in section 10.8. Some of the output has been deleted for the sake of clarity. The entire output file is about 1100 lines long.

Loading hadd.bm

Note that (NUMBERP (XORGATE–SPEC IN1 IN2)) is a theorem.

[0.0 0.0 0.0]

Note that (NUMBERP (ANDGATE2–SPEC IN1 IN2)) is a theorem.

[0.0 0.0 0.0]

From the definition we can conclude that (NUMBERP (HALFADDER–SPEC X Y)) is a theorem.

[0.0 0.0 0.0]

From the definition we can conclude that (NUMBERP (A1 X Y)) is a theorem.

[0.0 0.0 0.0]

From the definition we can conclude that (NUMBERP (X1 X Y)) is a theorem.

[0.0 0.0 0.0]

From the definition we can conclude that (BVP (HALFADDER–CIRCUIT X Y)) is a theorem.

[0.0 0.0 0.0]

This formula can be simplified, using the abbreviations AND, IMPLIES, and HALFADDER–SPEC, to:

(IMPLIES (AND (BVP X)
 (BVP Y)
 (EQUAL (BV–LENGTH X) 1)
 (EQUAL (BV–LENGTH Y) 1))
 (EQUAL (BV–TO–NAT (HALFADDER–CIRCUIT X Y))
 (PLUS (BV–TO–NAT X) (BV–TO–NAT Y))))),

which simplifies, appealing to the lemmas BV–FIX–BV–AND, BV–LENGTH–BV–AND, BV–TO–NAT–TO–BV, BV–FIX–BV–XOR, BV–LENGTH–BV–XOR, COMMUTATIVITY–OF–TIMES, and BV–TO–NAT–BV–APPEND, and unfolding A1, ANDGATE2–SPEC, EQUAL, X1, XORGATE–SPEC, HALFADDER–CIRCUIT, and EXP, to the conjecture:

(IMPLIES (AND (BVP X)

(BVP Y)

(EQUAL (BV-LENGTH X) 1)

(EQUAL (BV-LENGTH Y) 1))

(EQUAL (PLUS (BV-TO-NAT (BV-XOR X Y))

(TIMES 2 (BV-TO-NAT (BV-AND X Y))))

(PLUS (BV-TO-NAT X) (BV-TO-NAT Y))))).

Give the above formula the name *1.

We will appeal to induction. There are six plausible inductions.

However, they merge into one likely candidate induction. We will induct according to the following scheme:

(AND (IMPLIES (BV-NILP X) (p X Y))

(IMPLIES (AND (NOT (BV-NILP X))

(p (BV-VEC X) (BV-VEC Y)))

(p X Y))).

Linear arithmetic, the lemmas BV-VEC-LESSEQP and BV-VEC-LESSP, and the definition of BV-NILP can be used to prove that the measure (COUNT X) decreases according to the well-founded relation LESSP in each induction step of the scheme. Note, however, the inductive instance chosen for Y. The above induction scheme generates four new goals:

Case 4. (IMPLIES (AND (BV-NILP X)

(BVP X)

(BVP Y)

(EQUAL (BV-LENGTH X) 1)

(EQUAL (BV-LENGTH Y) 1))

(EQUAL (PLUS (BV-TO-NAT (BV-XOR X Y))

(TIMES 2 (BV-TO-NAT (BV-AND X Y))))

(PLUS (BV-TO-NAT X) (BV-TO-NAT Y))))),

which simplifies, unfolding BV-NILP, BVP, BV-LENGTH, and EQUAL, to:

T.

Case 3. (IMPLIES (AND (NOT (BV-NILP X))

(NOT (EQUAL (BV-LENGTH (BV-VEC X)) 1))

(BVP X)

(BVP Y)

(EQUAL (BV-LENGTH X) 1)
 (EQUAL (BV-LENGTH Y) 1))
 (EQUAL (PLUS (BV-TO-NAT (BV-XOR X Y))
 (TIMES 2 (BV-TO-NAT (BV-AND X Y))))
 (PLUS (BV-TO-NAT X) (BV-TO-NAT Y))))),

which simplifies, rewriting with the lemmas EQUAL-BV-LENGTH-0, ADD1-EQUAL,
 BV-NILP-BV-XOR, BV-TO-NAT-BV-NILP, BV-TO-NAT-BV, BV-NILP-BV-AND,
 COMMUTATIVITY-OF-PLUS, PLUS-1, and PLUS-STOPPER, and unfolding the functions
 BV-NILP, BV-LENGTH, NUMBERP, BV-XOR, B-XOR, TIMES, PLUS, BV-AND, B-AND,
 BV-TO-NAT, and ZEROP, to 12 new conjectures:

Case 3.12.

(IMPLIES (AND (NOT (EQUAL X (BV-NIL)))
 (NOT (EQUAL (BV-LENGTH (BV-VEC X)) 1))
 (BVP X)
 (BVP Y)
 (EQUAL (BV-VEC X) (BV-NIL))
 (NOT (EQUAL Y (BV-NIL)))
 (EQUAL (ADD1 (BV-LENGTH (BV-VEC Y)))
 1)
 (NOT (TRUEP (BV-BIT Y)))
 (NOT (TRUEP (BV-BIT X)))
 (BV-BIT X)
 (NOT (BV-BIT Y)))
 (EQUAL (PLUS (TIMES 2 0) 1)
 (PLUS (TIMES 2 (BV-TO-NAT (BV-VEC X))
 (TIMES 2 (BV-TO-NAT (BV-VEC Y))))))),

which again simplifies, trivially, to:

T.

The other subcases of case 3 are dealt with in a similar manner – this part of the output is
 deleted for the sake of clarity.

Case 3.1.

(IMPLIES (AND (NOT (EQUAL X (BV-NIL)))
 (NOT (EQUAL (BV-LENGTH (BV-VEC X)) 1))

(BVP X)
 (BVP Y)
 (EQUAL (BV-VEC X) (BV-NIL))
 (NOT (EQUAL Y (BV-NIL)))
 (EQUAL (ADD1 (BV-LENGTH (BV-VEC Y)))
 1)
 (TRUEP (BV-BIT Y))
 (TRUEP (BV-BIT X))
 (NOT (BV-BIT X)))
 (EQUAL (PLUS (TIMES 2 0) 1)
 (PLUS (ADD1 (TIMES 2 (BV-TO-NAT (BV-VEC X))))
 (ADD1 (TIMES 2 (BV-TO-NAT (BV-VEC Y)))))),

which again simplifies, trivially, to:

T.

Case 2. (IMPLIES (AND (NOT (BV-NILP X))
 (NOT (EQUAL (BV-LENGTH (BV-VEC Y)) 1))
 (BVP X)
 (BVP Y)
 (EQUAL (BV-LENGTH X) 1)
 (EQUAL (BV-LENGTH Y) 1))
 (EQUAL (PLUS (BV-TO-NAT (BV-XOR X Y))
 (TIMES 2 (BV-TO-NAT (BV-AND X Y))))
 (PLUS (BV-TO-NAT X) (BV-TO-NAT Y)))).

This simplifies, applying EQUAL-BV-LENGTH-0, ADD1-EQUAL, BV-NILP-BV-XOR,
 BV-TO-NAT-BV-NILP, BV-TO-NAT-BV, BV-NILP-BV-AND, COMMUTATIVITY-OF-PLUS,
 PLUS-1, and PLUS-STOPPER, and expanding the functions BV-NILP, BV-LENGTH,
 NUMBERP, BV-XOR, B-XOR, TIMES, PLUS, BV-AND, B-AND, BV-TO-NAT, and ZEROP, to
 12 new formulas:

Case 2.12.

(IMPLIES (AND (NOT (EQUAL X (BV-NIL)))
 (NOT (EQUAL (BV-LENGTH (BV-VEC Y)) 1))
 (BVP X)
 (BVP Y)
 (EQUAL (BV-VEC X) (BV-NIL))

```

(NOT (EQUAL Y (BV-NIL)))
(EQUAL (ADD1 (BV-LENGTH (BV-VEC Y)))
  1)
(NOT (TRUEP (BV-BIT Y)))
(NOT (TRUEP (BV-BIT X)))
(BV-BIT X)
(NOT (BV-BIT Y)))
(EQUAL (PLUS (TIMES 2 0) 1)
  (PLUS (TIMES 2 (BV-TO-NAT (BV-VEC X)))
    (TIMES 2 (BV-TO-NAT (BV-VEC Y))))),

```

which again simplifies, clearly, to:

T.

The other subcases of case 2 are dealt with in a similar manner – this part of the output is deleted for the sake of clarity.

Case 2.1.

```

(IMPLIES (AND (NOT (EQUAL X (BV-NIL)))
  (NOT (EQUAL (BV-LENGTH (BV-VEC Y)) 1))
  (BVP X)
  (BVP Y)
  (EQUAL (BV-VEC X) (BV-NIL))
  (NOT (EQUAL Y (BV-NIL)))
  (EQUAL (ADD1 (BV-LENGTH (BV-VEC Y)))
    1)
  (TRUEP (BV-BIT Y))
  (TRUEP (BV-BIT X))
  (NOT (BV-BIT X)))
  (EQUAL (PLUS (TIMES 2 0) 1)
    (PLUS (ADD1 (TIMES 2 (BV-TO-NAT (BV-VEC X)))
      (ADD1 (TIMES 2 (BV-TO-NAT (BV-VEC Y))))))),

```

which again simplifies, obviously, to:

T.

Case 1. (IMPLIES

```

(AND (NOT (BV-NILP X))

```

```

(EQUAL (PLUS (BV-TO-NAT (BV-XOR (BV-VEC X) (BV-VEC Y)))
(TIMES 2
(BV-TO-NAT (BV-AND (BV-VEC X) (BV-VEC Y)))))
(PLUS (BV-TO-NAT (BV-VEC X))
(BV-TO-NAT (BV-VEC Y)))))
(BVP X)
(BVP Y)
(EQUAL (BV-LENGTH X) 1)
(EQUAL (BV-LENGTH Y) 1))
(EQUAL (PLUS (BV-TO-NAT (BV-XOR X Y))
(TIMES 2 (BV-TO-NAT (BV-AND X Y)))))
(PLUS (BV-TO-NAT X) (BV-TO-NAT Y)))).

```

This simplifies, rewriting with the lemmas H-TIM, TIMES-1, EQUAL-BV-LENGTH-0, ADD1-EQUAL, BV-NILP-BV-XOR, BV-TO-NAT-BV-NILP, BV-TO-NAT-BV, BV-NILP-BV-AND,

COMMUTATIVITY-OF-PLUS, PLUS-1, and PLUS-STOPPER, and unfolding the definitions of BV-NILP, SUB1, NUMBERP, EQUAL, TIMES, BV-LENGTH, BV-XOR, B-XOR, PLUS, BV-AND, B-AND, BV-TO-NAT, and ZEROP, to the following 12 new conjectures:

Case 1.12.

```

(IMPLIES
(AND
(NOT (EQUAL X (BV-NIL)))
(EQUAL (PLUS (BV-TO-NAT (BV-XOR (BV-VEC X) (BV-VEC Y)))
(TIMES 2
(BV-TO-NAT (BV-AND (BV-VEC X) (BV-VEC Y)))))
(PLUS (BV-TO-NAT (BV-VEC X))
(BV-TO-NAT (BV-VEC Y)))))
(BVP X)
(BVP Y)
(EQUAL (BV-VEC X) (BV-NIL))
(NOT (EQUAL Y (BV-NIL)))
(EQUAL (ADD1 (BV-LENGTH (BV-VEC Y)))
1)
(NOT (TRUEP (BV-BIT Y)))

```

```

(NOT (TRUEP (BV-BIT X)))
(BV-BIT X)
(NOT (BV-BIT Y)))
(EQUAL (PLUS (TIMES 2 0) 1)
  (PLUS (TIMES 2 (BV-TO-NAT (BV-VEC X)))
    (TIMES 2 (BV-TO-NAT (BV-VEC Y)))))).

```

This again simplifies, trivially, to:

T.

The other subcases of case 1 are dealt with in a similar manner – this part of the output is again deleted for the sake of clarity.

Case 1.1.

```

(IMPLIES
  (AND
    (NOT (EQUAL X (BV-NIL)))
    (EQUAL (PLUS (BV-TO-NAT (BV-XOR (BV-VEC X) (BV-VEC Y)))
      (TIMES 2
        (BV-TO-NAT (BV-AND (BV-VEC X) (BV-VEC Y)))))
      (PLUS (BV-TO-NAT (BV-VEC X))
        (BV-TO-NAT (BV-VEC Y)))))
    (BVP X)
    (BVP Y)
    (EQUAL (BV-VEC X) (BV-NIL))
    (NOT (EQUAL Y (BV-NIL)))
    (EQUAL (ADD1 (BV-LENGTH (BV-VEC Y)))
      1)
    (TRUEP (BV-BIT Y))
    (TRUEP (BV-BIT X))
    (NOT (BV-BIT X)))
    (EQUAL (PLUS (TIMES 2 0) 1)
      (PLUS (ADD1 (TIMES 2 (BV-TO-NAT (BV-VEC X))))
        (ADD1 (TIMES 2 (BV-TO-NAT (BV-VEC Y)))))).

```

This again simplifies, clearly, to:

T.

That finishes the proof of *1. Q.E.D.

[0.0 66.3 3.7]

Finished loading hadd.bm

A.6. Transformer output

The final STRICT code generated by the Transformer (Chapter 11) is as follows.

```
BLOCK correctable(n: integer)
  HAVING (a:posint[n] i:posint[n] din,clk,res: WIRE):
    (out:bit dout:WIRE)
  SIZE 1 BY area
  INTENDED BEHAVIOUR
    WHENEVER
      change(clk):
        WITHIN (time)
          SET
            output=nil ;
  USE STRUCTURE
  {
    INSTANCE
      and21, and22: and2 (n)
      cmp1, cmp2, cmp3: cmp (n)
      eval0, eval1, eval2, eval3: eval (n)
      alfapow1, alfapow2, alfapow3: alfapow (n)
      mul1, mul2, mul3: mul (n)
      d1: reg (n)
    USING
      eval1(a, 0, din)
      mul1(1, i)
      alfapow1(mul1.out, eval0.out, eval2.dout)
      eval0(a, 1, din)
      eval4(a, 0, din)
      mul2(2, i)
      alfapow2(mul2.out, eval0.out, eval0.dout)
      eval2(a, 2, din)
      mul3(3, i)
```



```

        alfapow3(mul3.out,eval0.out,eval0.dout)
        cmp3(eval3.out,alfapow3.out)
        cmp2(eval2.out,alfapow2.out)
        and22(cmp2.out,cmp3.out)
        cmp1(eval1.out,alfapow1.out)
        d1(alfapow2.dout,clk,res)
        and21(cmp1.out,and22.out)

    MAKE

        correctable:= and21.out
        dout := d1.dout
    }

END

BLOCK eval(n: integer)
    HAVING (a:posint[n] j:posint[n] din,clk,res: WIRE):
        (out:posint[n] dout:WIRE)

    SIZE 1 BY area
    INTENDED BEHAVIOUR

    WHENEVER
        change(clk):
            WITHIN (time)
                SET
                    output=nil ;

    USE STRUCTURE
    {
        INSTANCE
            newsum1:newsum(n)
            d1:reg(n)

        USING
            d1(din,clk,res)
            newsum1(31,0,a,j,d1.out)

        MAKE
            eval:= newsum1.out

```

```

        dout := newsum1.dout
    }
END
BLOCK newsum(n: integer)
    HAVING (i:posint[n] g:posint[n] a:posint[n]
            j:posint[n] din,clk,res: WIRE):
        (out:posint[n] dout:WIRE)
    SIZE 1 BY area
    INTENDED BEHAVIOUR
        WHENEVER
            change(clk):
                WITHIN (time)
                    SET
                        output=nil ;
    USE STRUCTURE
    {
        INSTANCE
            cmp1:cmp(n)
            sub1:sub(n)
            reg1,reg2:reg(n)
            mux1,mux2:mux(n)
            exn1:exn(n)
            alfapow1:alfapow(n)
            regc1:reg(n)
            muxc1:mux(n)
            d1:reg(n)
        USING
            mux1(cmp1.out,i,sub1.out)
            reg1(mux1.out,muxc1.out,res)
            mux2(cmp1.out,g,exn1.out)
            reg2(mux2.out,muxc1.out,res)
            alfapow1(j,reg2.out)
    }

```

```

        exn1(a,alfapow1.out)
        sub1(reg1.out)
        cmp1(reg1.out,0)
        d1(regc2.out,clk,res)
        muxc1(cmp1.out,d1.out,din)
        regc1(muxc1.out,clk,res)

    MAKE

        newsum ::= reg2.out
        dout ::= cmp1.out
    }

END

BLOCK alfapow(n: integer)
    HAVING (n:posint[n] g:posint[n] din,clk,res: WIRE):
        (out:posint[n] dout:WIRE)

    SIZE 1 BY area

    INTENDED BEHAVIOUR

        WHENEVER

            change(clk):

                WITHIN (time)

                    SET

                        output=nil ;

USE STRUCTURE

{
    INSTANCE

        cmp1:cmp(n)

        sub1:sub(n)

        reg1,reg2:reg(n)

        mux1,mux2:mux(n)

        alfaxv1:alfaxv(n)

        regc1:reg(n)

        muxc1:mux(n)

        d1:reg(n)

```

USING

```
mux1(cmp1.out,n,sub1.out)
reg1(mux1.out,muxc1.out,res)
mux2(cmp1.out,g,alfaxv1.out)
reg2(mux2.out,muxc1.out,res)
alfaxv1(reg2.out)
sub1(reg1.out)
cmp1(reg1.out,0)
d1(regc1.out,clk,res)
muxc1(cmp1.out,d1.out,din)
regc1(muxc1.out,clk,res)
```

MAKE

```
alfapow:= reg2.out
dout := cmp1.out
```

}

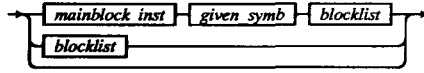
END

B. STRICT SYNTAX

complete_descrip



strict_descript



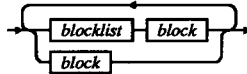
mainblock_inst



start_symb



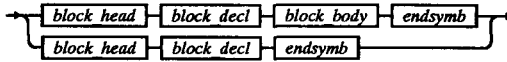
blocklist



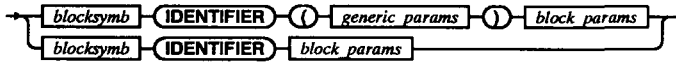
given_symb



block



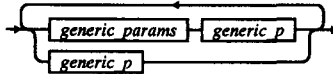
block_head



blocksymb



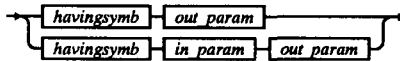
generic_params

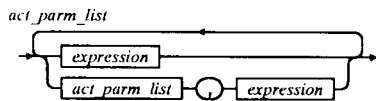
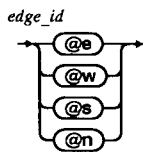
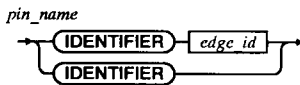
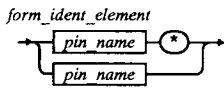
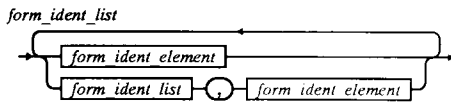
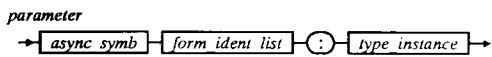
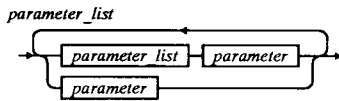
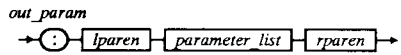
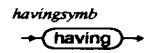


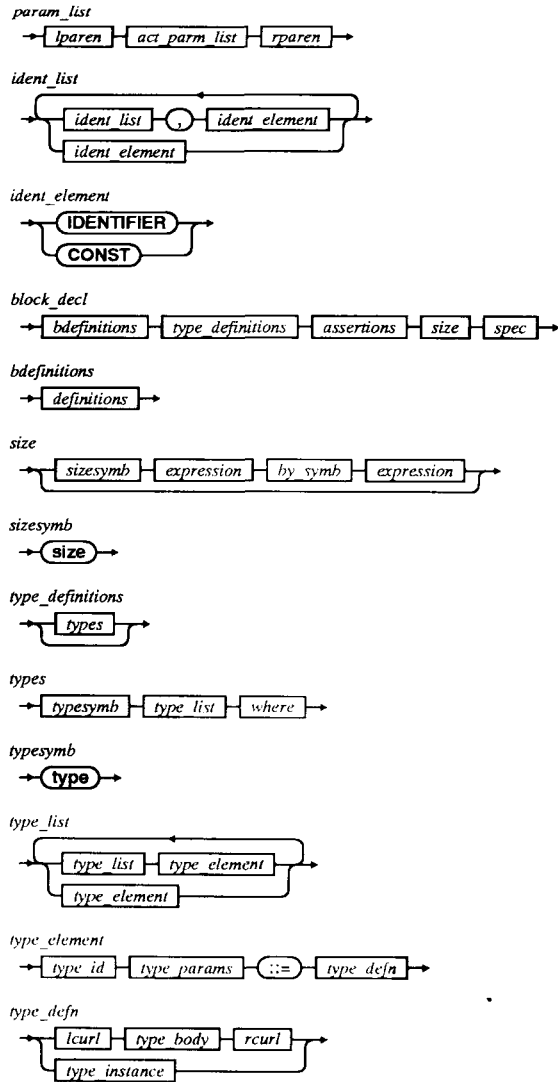
generic_p



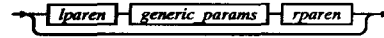
block_params



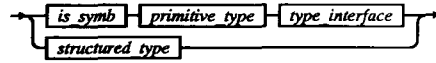




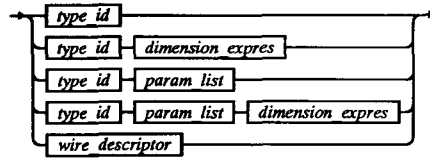
type_params



type_body



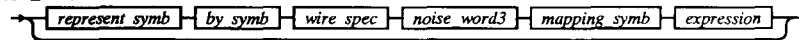
type_instance



is_symb



type_interface



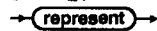
by_symb



mapping_symb



represent_symb



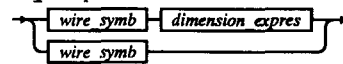
wire_spec



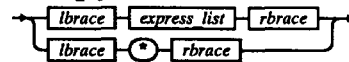
wire_symb

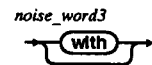
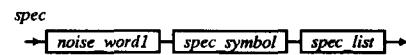
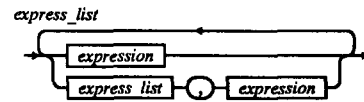
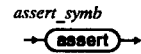
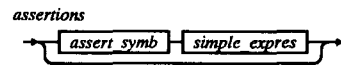
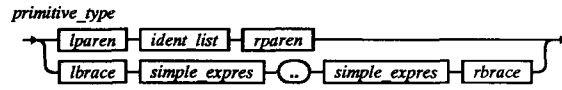
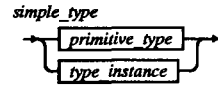
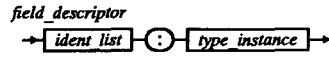
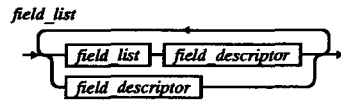
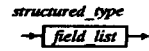


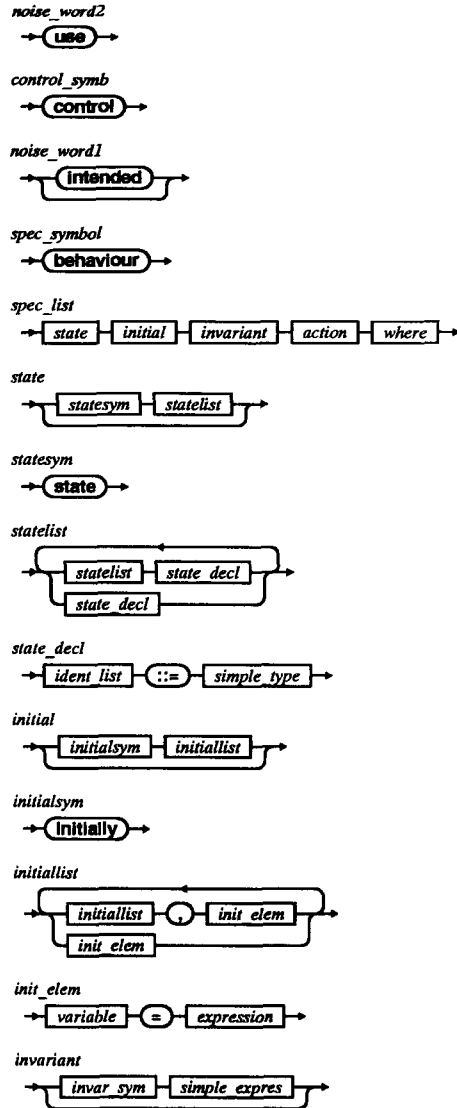
wire_descriptor

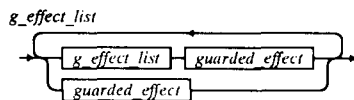
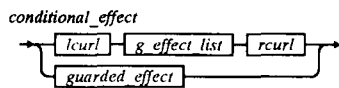
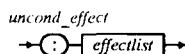
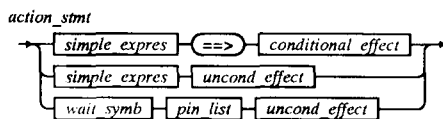
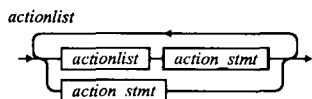
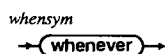
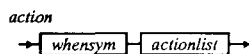
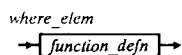
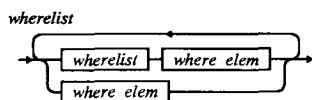
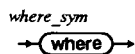
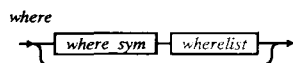
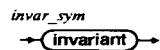


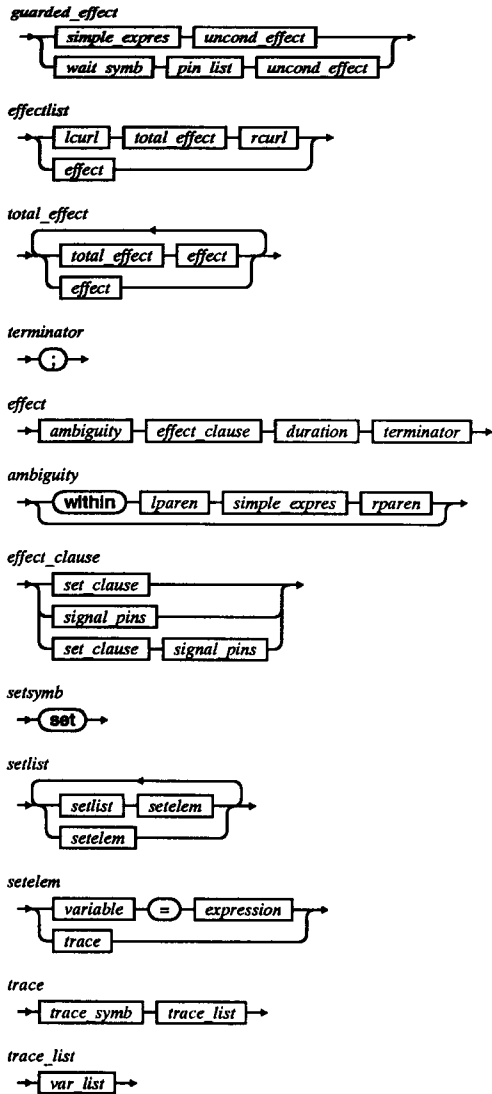
dimension_expres

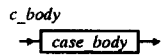
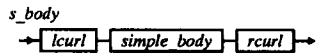
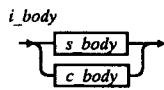
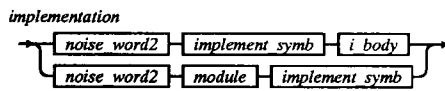
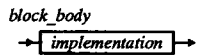
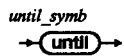
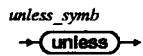
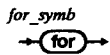
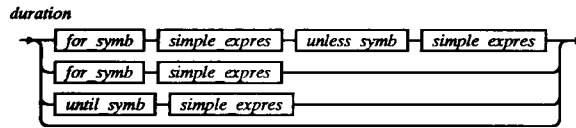


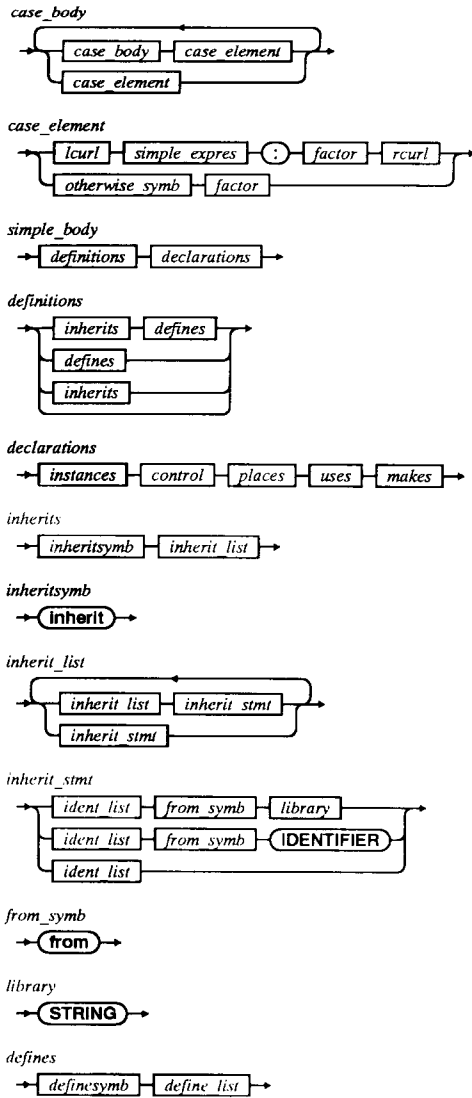


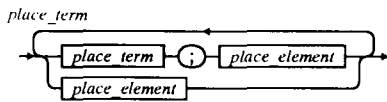
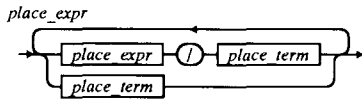
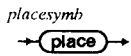
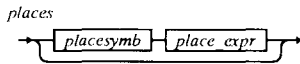
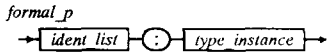
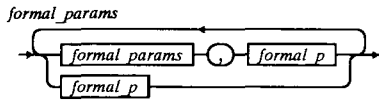
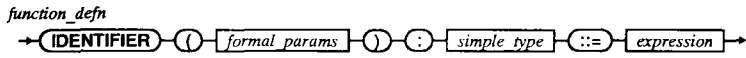
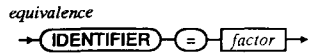
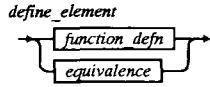
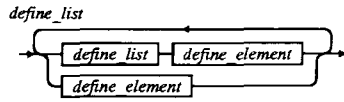
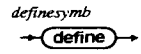


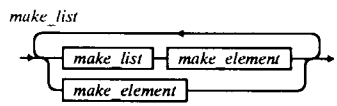
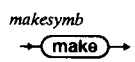
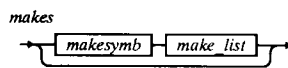
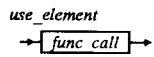
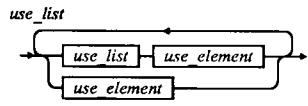
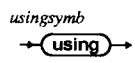
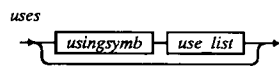
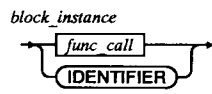
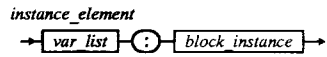
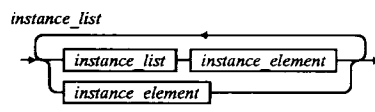
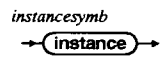
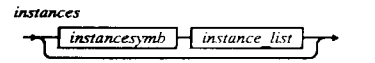








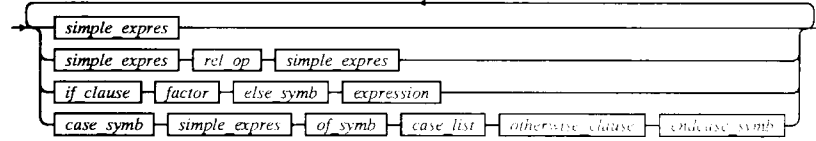




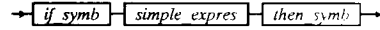
make_element



expression



if_clause



if_symb



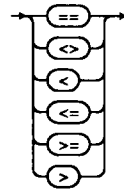
then_symb



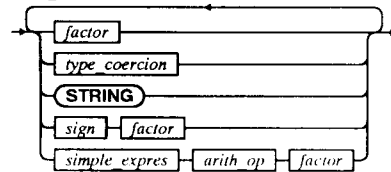
else_symb



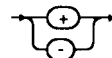
rel_op

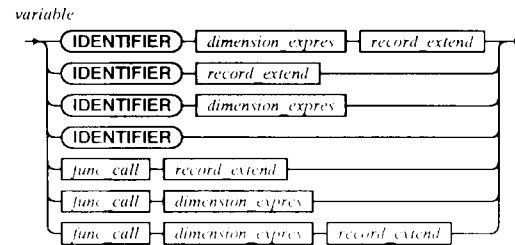
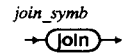
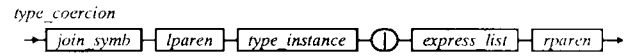
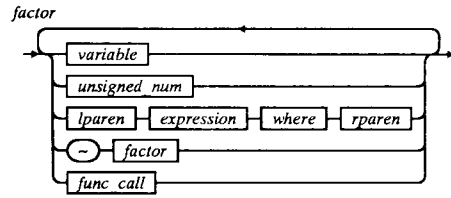
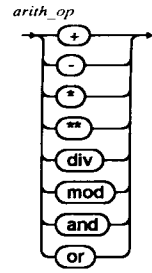


simple_expres



sign

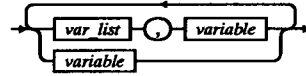




record_extend



var_list



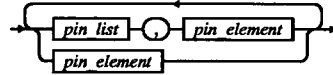
unsigned_num



endsymb



pin_list



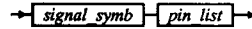
pin_element



wait_symb



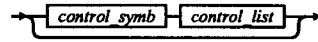
signal_pins



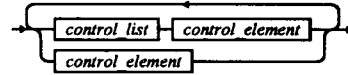
signal_symb



control



control_list



control_element



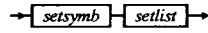
signal_section



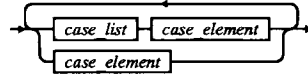
signal_clause



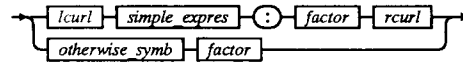
set_clause



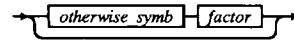
case_list



case_element



otherwise_clause



of_symb



otherwise_symb



endcase_symb



module



async_symb

